The Python Library Reference

Release 3.7.17

Guido van Rossum
and the Python development team

junho 28, 2023

Python Software Foundation
Email: docs@python.org

Sumário

Enquanto reference-index descreve a sintaxe e a semântica exatas da linguagem Python, este manual de referência de bibliotecas descreve a biblioteca padrão que é distribuída com o Python. Ele também descreve alguns dos componentes opcionais que são comumente incluídos nas distribuições do Python.

A biblioteca padrão do Python é muito extensa, oferecendo uma ampla gama de recursos, conforme indicado pelo longo índice listado abaixo. A biblioteca contém módulos internos (escritos em C) que fornecem acesso à funcionalidade do sistema, como E/S de arquivos que de outra forma seriam inacessíveis para programadores Python, bem como módulos escritos em Python que fornecem soluções padronizadas para muitos problemas que ocorrem em programação cotidiana. Alguns desses módulos são explicitamente projetados para incentivar e aprimorar a portabilidade de programas em Python, abstraindo os detalhes da plataforma em APIs neutras em plataforma.

Os instaladores do Python para a plataforma Windows geralmente incluem toda a biblioteca padrão e muitas vezes também incluem muitos componentes adicionais. Para sistemas operacionais semelhantes a Unix, o Python é normalmente fornecido como uma coleção de pacotes, portanto, pode ser necessário usar as ferramentas de empacotamento fornecidas com o sistema operacional para obter alguns ou todos os componentes opcionais.

Além da biblioteca padrão, há uma coleção crescente de vários milhares de componentes (de programas e módulos individuais a pacotes e frameworks de desenvolvimento de aplicativos inteiros), disponíveis no Python Package Index.

CAPÍTULO 1

Introdução

A “biblioteca Python” contém vários tipos diferentes de componentes.

Ela contém tipos de dados que seriam normalmente considerados como parte “central” de uma linguagem, tais como números e listas. Para esses tipos, o núcleo da linguagem Python define a forma de literais e coloca algumas restrições em suas semânticas, mas não define completamente as semânticas. (Por outro lado, o núcleo da linguagem define propriedades sintáticas como a ortografia e a prioridade de operadores.)

A biblioteca também contém funções e exceções prontas — objetos que podem ser usados por todo o código Python sem a necessidade de uma instrução `import`. Alguns desses são definidos pelo núcleo da linguagem, mas muitos não são essenciais para as semânticas principais e são apenas descritos aqui.

A maior parte da biblioteca, entretanto, consiste em uma coleção de módulos. Há muitas formas de dissecar essa coleção. Alguns módulos são escritos em C e colocados no interpretador do Python; outros são escritos em Python e importados na forma de código. Alguns módulos fornecem interfaces que são muito específicas do Python, como imprimir um rastreamento de pilha; alguns fornecem interfaces que são específicas para um sistema operacional em particular, tais como acessar hardware específico; outros fornecem interfaces que são específicas de um domínio de aplicação em particular, como a World Wide Web. Alguns módulos estão disponíveis em todas as versões do Python; outros estão apenas disponíveis quando o sistema subjacente suporta ou necessita deles; e ainda outros estão disponíveis apenas quando uma opção de configuração em particular foi escolhida no momento em que o Python foi compilado e instalado.

Este manual está organizado “de dentro para fora”: ele primeiro descreve as funções inclusas, tipos de dados e exceções, e finalmente os módulos, agrupados em capítulos de módulos relacionados.

Isto significa que, se você começar a ler este manual do início, e pular para o próximo capítulo quando estiver entediado, você terá uma visão geral razoável dos módulos disponíveis e áreas de aplicação que são suportadas pela biblioteca Python. É claro, você não *tem* que ler como se fosse um romance — você também pode navegar pela tabela de conteúdos (no início do manual), ou procurar por uma função, módulo ou termo específicos no índice (na parte final). E finalmente, se você gostar de aprender sobre assuntos diversos, você pode escolher um número de página aleatório (veja module `random`) e leia uma seção ou duas. Independente da ordem na qual você leia as seções deste manual, ajuda iniciar pelo capítulo *Funções embutidas*, já que o resto do manual requer familiaridade com este material.

E que o show comece!

1.1 Observações sobre disponibilidade

- Uma observação “Disponibilidade: Unix” significa que essa função é comumente encontrada em sistemas Unix. Não faz nenhuma reivindicação sobre sua existência em um sistema operacional específico.
- Se não for observado separadamente, todas as funções que afirmam “Disponibilidade: Unix” são suportadas no Mac OS X, que é baseado em um núcleo Unix.

CAPÍTULO 2

Funções embutidas

O interpretador do Python possui várias funções e tipos embutidos que sempre estão disponíveis. A seguir listamos todas as funções em ordem alfabética.

		Funções embutidas		
abs()	*delattr()*	*hash()*	*memoryview()*	*set()*
all()	*dict()*	*help()*	*min()*	*setattr()*
any()	*dir()*	*hex()*	*next()*	*slice()*
ascii()	*divmod()*	*id()*	*object()*	*sorted()*
bin()	*enumerate()*	*input()*	*oct()*	*staticmethod()*
bool()	*eval()*	*int()*	*open()*	*str()*
breakpoint()	*exec()*	*isinstance()*	*ord()*	*sum()*
bytearray()	*filter()*	*issubclass()*	*pow()*	*super()*
bytes()	*float()*	*iter()*	*print()*	*tuple()*
callable()	*format()*	*len()*	*property()*	*type()*
chr()	*frozenset()*	*list()*	*range()*	*vars()*
classmethod()	*getattr()*	*locals()*	*repr()*	*zip()*
compile()	*globals()*	*map()*	*reversed()*	*__import__()*
complex()	*hasattr()*	*max()*	*round()*	

abs(*x*)
Retorna o valor absoluto de um número. O argumento pode ser um inteiro ou um número de ponto flutuante. Se o argumento é um número complexo, sua magnitude é retornada.

all(*iterable*)
Retorna `True` se todos os elementos de *iterable* são verdadeiros (ou se *iterable* estiver vazio). Equivalente a:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

any(*iterable*)

Retorna `True` se alguma elemento de *iterable* for verdadeiro. Se *iterable* estiver vazio, retorna `False`. Equivalente a:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

ascii(*object*)

Como *repr()*, retorna uma string contendo uma representação imprimível de um objeto, mas faz escape de caracteres não-ASCII na string retornada por *repr()* usando sequências de escapes `\x`, `\u` or `\U`. Isto gera uma string similar ao que é retornado por *repr()* no Python 2.

bin(*x*)

Converte um número inteiro para uma string de binários prefixada com "0b". O resultado é uma expressão Python válida. Se *x* não é um objeto Python *int*, ele tem que definir um método `__index__()` que devolve um inteiro. Alguns exemplos:

```
>>> bin(3)
'0b11'
>>> bin(-10)
'-0b1010'
```

Se o prefixo "0b" é desejado ou não, você pode usar uma das seguintes maneiras.

```
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

Veja também *format()* para mais informações.

class bool([*x*])

Retorna um valor Booleano, isto é, `True` ou `False`. *x* é convertida usando o *procedimento de teste verdade padrão*. Se *x* é falso ou foi omitido, isso retorna `False`; senão ele retorna `True`. A classe *bool* é uma subclasse de *int* (veja *Tipos Numéricos — int, float, complex*). Ela não pode ser usada para criar outra subclasse. Suas únicas instâncias são `False` e `True` (veja *Valores Booleanos*).

Alterado na versão 3.7: *x* é agora um parâmetro somente posicional.

breakpoint(**args*, ***kws*)

Esta função coloca você no depurador no local da chamada. Especificamente, ela chamada *sys.breakpointhook()*, passando `args` and `kws` diretamente. Por padrão, `sys.breakpointhook()` chama *pdb.set_trace()* não esperando nenhum argumento. Neste caso, isso é puramente uma função de conveniência para você não precisar importar *pdb* explicitamente ou digitar mais código para entrar no depurador. Contudo, *sys.breakpointhook()* pode ser configurado para alguma outra função e *breakpoint()* irá automaticamente chamá-la, permitindo você ir para o depurador de sua escolha.

Novo na versão 3.7.

class bytearray([*source*[, *encoding*[, *errors*]]])

Retorna um novo vetor de bytes. A classe *bytearray* é uma sequência mutável de inteiros no intervalo 0 <= x < 256. Ela tem a maior parte dos métodos mais comuns de sequências mutáveis, descritas em *Tipos de Sequências Mutáveis*, assim como a maior parte dos métodos que o tipo *bytes* tem, veja *Operações com Bytes e Bytearray*.

O parâmetro opcional *source* pode ser usado para inicializar o vetor de algumas maneiras diferentes:

- Se é uma *string*, você deve informar o parâmetro *encoding* (e opcionalmente, *errors*); `bytearray()` então converte a string para bytes usando `str.encode()`.
- Se é um *inteiro*, o vetor terá esse tamanho e será inicializado com bytes nulos.
- Se é um objeto em conformidade com a interface *buffer*, um buffer de objeto somente leitura será usado para inicializar o vetor de bytes.
- Se é um *iterável*, deve ser um iterável de inteiros no intervalo `0 <= x < 256`, que serão usados como o conteúdo inicial do vetor.

Sem nenhum argumento, um vetor de tamanho 0 é criado.

Veja também *Tipos de Sequência Binária — bytes, bytearray, memoryview* and *Bytearray Objects*.

class bytes([*source*[, *encoding*[, *errors*]]])

Retorna um novo objeto "bytes", que é uma sequência imutável de inteiros no intervalo"0 <= x < 256". `bytes` é uma versão imutável de `bytearray` – tem os mesmos métodos de objetos imutáveis e o mesmo comportamento de índices e fatiamento.

Consequentemente, argumentos do construtor são interpretados como os de `bytearray()`.

Objetos bytes também podem ser criados com literais, veja strings.

Veja também *Tipos de Sequência Binária — bytes, bytearray, memoryview*, *Objetos Bytes*, e *Operações com Bytes e Bytearray*.

callable(*object*)

Devolve `True` se o argumento *object* parece chamável, `False` caso contrário. Se devolve `True`, ainda é possível que a chamada falhe, mas se é `False`, chamar *object* nunca será bem sucedido. Note que classes são chamáveis (chamar uma classe devolve uma nova instância); instâncias são chamáveis se suas classes possuem um método `__call__()`.

Novo na versão 3.2: Esta função foi removida na versão 3.0, mas retornou no Python 3.2.

chr(*i*)

Retorna o caractere que é apontado pelo inteiro *i* no código Unicode. Por exemplo, `chr(97)` retorna a string `'a'`, enquanto `chr(8364)` retorna a string `'€'`. É o inverso de `ord()`.

O intervalo válido para o argumento vai de 0 até 1,114,111 (0x10FFFF na base 16). Será lançada uma exceção `ValueError` se *i* estiver fora desse intervalo.

@classmethod

Transforma um método em um método de classe.

Um método de classe recebe a classe como primeiro argumento implícito, exatamente como uma método de instância recebe a instância. Para declarar um método de classe, faça dessa forma:

```
class C:
    @classmethod
    def f(cls, arg1, arg2, ...): ...
```

O termo `@classmethod` é uma função *decoradora* – veja function para detalhes.

Um método de classe pode ser chamado tanto da classe (como em `C.f()`) quanto da instância (como em `C().f()`). A instância é ignorada, exceto por sua classe. Se um método de classe é chamado por uma classe derivada, o objeto da classe derivada é passado como primeiro argumento implícito.

Métodos de classe são diferentes de métodos estáticos em C++ ou Java. Se você quer saber desses, veja `staticmethod()`.

Para mais informações sobre métodos de classe, veja types.

compile(*source*, *filename*, *mode*, *flags=0*, *dont_inherit=False*, *optimize=-1*)

Compila o argumento *source* em código ou objeto AST. Objetos código podem ser executados por `exec()` ou `eval()`. *source* pode ser uma string normal, uma string byte, ou um objeto AST. Consulte a documentação do módulo `ast` para saber como trabalhar com objetos AST.

O argumento *filename* deve ser o arquivo de onde o código será lido; passe algum valor reconhecível se isso não foi lido de um arquivo (`'<string>'` é comumente usado).

O argumento *mode* especifica qual o tipo de código deve ser compilado; pode ser `'exec'` se *source* consiste de uma sequência de instruções, `'eval'` se consiste de uma única expressão, ou `'single'` se consiste de uma única instrução interativa (neste último caso, instruções que são avaliadas para alguma coisa diferente de `None` serão exibidas).

Os argumentos opcionais *flags* e *dont_inherit* controlam qual instrução futura afeta a compilação de *source*. Se nenhum está presente (ou ambos são zero) o código é compilado com as instruções futuras que estão agindo no código que está chamando `compile()`. If the *flags* argument is given and *dont_inherit* is not (or is zero) then the future statements specified by the *flags* argument are used in addition to those that would be used anyway. If *dont_inherit* is a non-zero integer then the *flags* argument is it – the future statements in effect around the call to compile are ignored.

Instruções futuras são especificadas por bits, assim pode ocorrer uma operação *OU* bit a bit para especificar múltiplas instruções. O sinalizador necessário para especificar um dado recurso pode ser encontrada no atributo `compiler_flag` na instância `_Feature` do módulo `__future__` module.

O argumento *optimize* especifica o nível de otimização do compilador; o valor padrão de `-1` seleciona o nível de otimização do interpretador dado pela opção `-O`. Níveis explícitos são `0` (nenhuma otimização; `__debug__` é verdadeiro), `1` (instruções `asserts` são removidas, `__debug__` é falso) ou `2` (strings de documentação também são removidas).

Essa função levanta `SyntaxError` se o código para compilar é inválido, e `ValueError` se o código contém bytes nulos.

Se você quer analisar código Python em sua representação AST, veja `ast.parse()`.

Nota: Quando compilindo uma string com código multi-linhas em modo `'single'` ou `'eval'`, entrada deve ser terminada por ao menos um caractere de nova linhas. Isso é para facilitar a detecção de instruções completas e incompletas no módulo `code`.

Aviso: É possível quebrar o interpretador Python com uma string suficiente grande/complexa quando compilando para uma objeto AST, devido limitações do tamanho da pilha no compilador AST do Python.

Alterado na versão 3.2: Permitido uso de marcadores de novas linhas no estilo Windows e Mac. Além disso, em modo `'exec'` a entrada não precisa mais terminar com uma nova linha. Também foi adicionado o parâmetro *optimize*.

Alterado na versão 3.5: Anteriormente, `TypeError` era levantada quando havia bytes nulos em *source*.

class complex([*real*[, *imag*]])

Retorna um número completo com o valor *real* + *imag**1j ou converte uma string ou número para um número complexo. Se o primeiro parâmetro é uma string, ele será interpretado como um número complexo e a função deve ser chamada sem um segundo parâmetro. O segundo parâmetro nunca deve ser uma string. Cada argumento pode ser qualquer tipo numérico (incluindo complexo). Se *imag* é omitido, seu valor padrão é zero e a construção funciona como uma conversão numérica, similar a `int` e `float`. Se os dois argumentos são omitidos, retorna `0j`.

Nota: Quando convertendo a partir de uma string, a string não pode conter espaços em branco em torno + central ou do operador −. Por exemplo, `complex('1+2j')` funciona, mas `complex('1 + 2j')` levanta *ValueError*.

O tipo complexo está descrito em *Tipos Numéricos — int, float, complex*.

Alterado na versão 3.6: Agrupar dígitos com sublinhados como em literais de código é permitido.

delattr(*object*, *name*)
Essa função está relacionada com *setattr()*. Os argumentos são um objeto e uma string. A string deve ser o nome de um dos atributos do objeto. A função remove o atributo indicado, desde que o objeto permita. Por exemplo, `delattr(x, 'foobar')` é equivalente a `del x.foobar`.

class dict(***kwarg*)
class dict(*mapping*, ***kwarg*)
class dict(*iterable*, ***kwarg*)
Cria um novo dicionário. O objeto *dict* é a classe do dicionário. Veja *dict* e *Tipo de Mapeamento — dict* para documentação sobre esta classe.

Para outros contêineres, consulte as classes internas *list*, *set* e *tuple*, bem como o módulo *collections*.

dir([*object*])
Sem argumentos, retorne a lista de nomes no escopo local atual. Com um argumento, tente retornar uma lista de atributos válidos para esse objeto.

Se o objeto tiver um método chamado `__dir__()`, esse método será chamado e deve retornar a lista de atributos. Isso permite que objetos que implementam uma função personalizada `__getattr__()` ou `__getattribute__()` personalizem a maneira *dir()* relata seus atributos.

Se o objeto não fornecer `__dir__()`, a função tentará o melhor possível para coletar informações do atributo *__dict__* do objeto, se definido, e do seu objeto de tipo. A lista resultante não está necessariamente completa e pode ser imprecisa quando o objeto possui um `__getattr__()` personalizado.

O mecanismo padrão *dir()* se comporta de maneira diferente com diferentes tipos de objetos, pois tenta produzir as informações mais relevantes e não completas:

- Se o objeto for um objeto de módulo, a lista conterá os nomes dos atributos do módulo.
- Se o objeto for um objeto de tipo ou classe, a lista conterá os nomes de seus atributos e recursivamente os atributos de suas bases.
- Caso contrário, a lista conterá os nomes dos atributos do objeto, os nomes dos atributos da classe e recursivamente os atributos das classes base da classe.

A lista resultante é alfabeticamente ordenada. Por exemplo:

```
>>> import struct
>>> dir()   # show the names in the module namespace
['__builtins__', '__name__', 'struct']
>>> dir(struct)   # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
>>> s = Shape()
```

(continua na próxima página)

(continuação da página anterior)

```
>>> dir(s)
['area', 'location', 'perimeter']
```

Nota: Como *dir()* é fornecido principalmente como uma conveniência para uso em um prompt interativo, ele tenta fornecer um conjunto interessante de nomes mais do que tenta fornecer um conjunto de nomes definido de forma rigorosa ou consistente, e seu comportamento detalhado pode mudar nos lançamentos. Por exemplo, os atributos de metaclasse não estão na lista de resultados quando o argumento é uma classe.

---

divmod(*a*, *b*)

Toma dois números (não complexos) como argumentos e retorne um par de números que consiste em seu quociente e restante ao usar a divisão inteira. Com tipos de operandos mistos, as regras para operadores aritméticos binários se aplicam. Para números inteiros, o resultado é o mesmo que `(a // b, a % b)`. Para números de ponto flutuante, o resultado é `(q, a % b)`, onde *q* geralmente é `math.floor(a / b)`, mas pode ser 1 a menos que isso. Em qualquer caso, `q * b + a % b` está muito próximo de *a*, se `a % b` é diferente de zero, tem o mesmo sinal que *b* e `0 <= abs(a % b) < abs(b)`.

enumerate(*iterable*, *start=0*)

Retorne um objeto enumerado. *iterable* deve ser uma sequência, um *iterador* ou algum outro objeto que suporte a iteração. O método *__next__()* do iterador retornado por *enumerate()* retorna uma tupla contendo uma contagem (de *start*, cujo padrão é 0) e os valores obtidos na iteração sobre *iterable*.

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

Equivalente a:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

eval(*expression*[, *globals*[, *locals*]])

Os argumentos são uma sequência de caracteres e globais e locais opcionais. Se fornecido, *globals* deve ser um dicionário. Se fornecido,locals* pode ser qualquer objeto de mapeamento.

The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* dictionaries as global and local namespace. If the *globals* dictionary is present and does not contain a value for the key __builtins__, a reference to the dictionary of the built-in module *builtins* is inserted under that key before *expression* is parsed. This means that *expression* normally has full access to the standard *builtins* module and restricted environments are propagated. If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed in the environment where *eval()* is called. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```
>>> x = 1
>>> eval('x+1')
2
```

Esta função também pode ser usada para executar objetos de código arbitrários (como os criados por *compile()*). Nesse caso, passe um objeto de código em vez de uma string. Se o objeto de código foi compilado com `'exec'`

como o argumento *mode*, o valor de retorno de `eval()` será' *None*'.

Dicas: a execução dinâmica de instruções é suportada pela função `exec()`. As funções `globals()` e `locals()` retornam o dicionário global e local atual, respectivamente, o que pode ser útil para ser usado por `eval()` ou `exec()`.

Veja `ast.literal_eval()` para uma função que pode avaliar com segurança strings com expressões contendo apenas literais.

exec(*object*[, *globals*[, *locals*]])

Esta função suporta execução dinâmica de código Python. O parâmetro *object* deve ser ou uma string ou um objeto contendo código. Se for uma string, a mesma é analisada como um conjunto de instruções Python, o qual é então executado (exceto caso um erro de sintaxe ocorra).[1] Se for um objeto com código, ele é simplesmente executado. Em todos os casos, espera-se que o código a ser executado seja válido como um arquivo de entrada (veja a seção "Arquivo de Entrada" no Manual de Referência). Tenha cuidado que as expressões `return` e `yield` não podem ser usadas fora das definições de funções mesmo dentro do contexto do código passado para a função `exec()` . O valor de retorno é sempre `None`.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only *globals* is provided, it must be a dictionary, which will be used for both the global and the local variables. If *globals* and *locals* are given, they are used for the global and local variables, respectively. If provided, *locals* can be any mapping object. Remember that at module level, globals and locals are the same dictionary. If exec gets two separate objects as *globals* and *locals*, the code will be executed as if it were embedded in a class definition.

Se o dicionário *globals* não contém um valor para a chave `__builtins__`, a referência para o dicionário do módulo embutido `builtins` é inserido com essa chave. A maneira que você pode controlar quais embutidos estão disponíveis para o código executado é inserindo seu próprio `__builtins__` dicionário em *globals* antes de passar para `exec()`.

Nota: As funções embutidas `globals()` e `locals()` retornam o dicionário global e local, respectivamente, o que pode ser útil para passar adiante e usar como segundo ou terceiro argumento para `exec()`.

Nota: *locals* padrão atua como descrito pela função `locals()` abaixo: modificações para o dicionário *locals* padrão não deveriam ser feitas. Se você precisa ver efeitos do código em *locals* depois da função `exec()` retornar passe um dicionário *locals* explícito.

---

filter(*function*, *iterable*)

Constrói um iterador a partir dos elementos de *iterable* para os quais *function* retorna true. *iterable* pode ser uma sequência (um container que suporta iteração) ou um iterador. Se *function* tiver o valor `None`, a função identidade é será usada, isto é, todos os elementos de *iterable* que retornam false são removidos.

Note que `filter(function, iterable)` é equivalente a expressão geradora `(item for item in iterable if function(item))` se function não tiver o valor `None` e `(item for item in iterable if item)` se function tiver o valor `None`.

Veja `itertools.filterfalse()` para a função complementar que retorna elementos de *iterable* para a qual *function* retorna false.

class float([*x*])

Retorna um número de ponto flutuante construído a partir de um número ou string * x *.

Se o argumento é uma string, ele deve conter um número decimal, opcionalmente precedido por um sinal, e opcionalmente possuir espaço em branco. O sinal opcional pode ser `'+'` ou `'-'`; um sinal de `'+'` não tem efeito no valor produzido. O argumento também pode ser uma string representando um NaN (indica que não é numero),

[1] Observe que o analisador aceita apenas a convenção de fim de linha no estilo Unix. Se você estiver lendo o código de um arquivo, use o modo de conversão de nova linha para converter novas linhas no estilo Windows ou Mac.

ou inifinito positivo/negativo. Mais precisamente, a entrada deve estar em conformidade com a seguinte gramática depois que caracteres em branco são removidos do início e do final da mesma:

```
sign           ::=  "+" | "-"
infinity       ::=  "Infinity" | "inf"
nan            ::=  "nan"
numeric_value  ::=  floatnumber | infinity | nan
numeric_string ::=  [sign] numeric_value
```

Aqui `floatnumber` é a forma literal de um ponto flutuante Python, descrito em floating. Caso isso não seja significante, então, por exemplo, "inf", "Inf", "INFINITY" e "iNfINity" são todas formas escritas válidas para infinito positivo.

Caso contrário, se o argumento é um inteiro ou um número de ponto flutuante, um número de ponto flutuante com o mesmo valor (com a precisão de ponto flutuante de Python) é devolvido. Se o argumento está fora do intervalo de um ponto flutuante Python, uma exceção *OverflowError* será lançada.

For a general Python object `x`, `float(x)` delegates to `x.__float__()`.

Se nenhum argumento for fornecido, `0.0` será retornado.

Exemplos:

```
>>> float('+1.23')
1.23
>>> float('   -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf
```

O tipo float é descrito em *Tipos Numéricos — int, float, complex*.

Alterado na versão 3.6: Agrupar dígitos com sublinhados como em literais de código é permitido.

Alterado na versão 3.7: *x* é agora um parâmetro somente posicional.

format(*value*[, *format_spec*])

Converte um valor *value* em uma representação "formatada", como controlado por *format_spec*. A interpretação de *format_spec* dependerá do tipo do argumento *value*, no entanto, há uma sintaxe de formatação padrão usada pela maioria dos tipos embutidos: formspecpec.

O padrão *format_spec* é uma string vazia que geralmente produz o mesmo efeito que chamar *str(valor)*.

Uma chamada de `format(value, format_spec)` é convertida em `type(value).__format__(value, format_spec)`, que ignora o dicionário da instância ao pesquisar o método `__format__()` do valor. A exceção *TypeError* é levantada se a pesquisa do método atingir *object* e o *format_spec* não estiver vazio, ou se o *format_spec* ou o valor de retorno não forem strings.

Alterado na versão 3.4: `object().__format__(format_spec)` levanta um *TypeError* se *format_spec* não for uma string vazia.

class frozenset([*iterable*])

Devolve um novo objeto *frozenset*, opcionalmente com elementos obtidos de *iterable*. `frozenset` é uma classe embutida. Veja *frozenset* e *Tipo Set — set, frozenset* para documentação sobre essas classes.

Para outros containers veja as classes embutidas *set*, *list*, *tuple*, e *dict*, bem como o módulo *collections*.

getattr(*object*, *name*[, *default*])
Devolve o valor do atributo *name* de *object*. *name* deve ser uma string. Se a string é o nome de um dos atributos do objeto, o resultado é o valor de tal atributo. Por exempleo, `getattr(x, 'foobar')` é equivalente a `x.foobar`. Se o atributo não existir, *default* é devolvido se tiver sido fornecido, caso contrário é levantada a exceção *AttributeError*.

globals()
Devolve um dicionário representando a tabela de símbolos global atual. É sempre o dicionário do módulo atual (dentro de uma função ou método, é o módulo onde está definido, não o módulo do qual é chamado).

hasattr(*object*, *name*)
Os argumentos são um objeto e uma string. O resultado é `True` se a string é o nome de um dos atributos do objeto, e `False` se ela não for. (Isto é implementado chamando `getattr(object, name)` e vendo se levanta um *AttributeError* ou não.)

hash(*object*)
Retorna o valor hash de um objeto (se houver um). Valores hash values são números inteiros. Eles são usados para rapidamente comparar chaves de dicionários durante uma pesquisa em um dicionário. Valores numéricos que ao serem comparados são iguais, possuem o mesmo valor hash (mesmo que eles sejam de tipos diferentes, como é o caso de 1 e 1.0).

Nota: Para objetos com métodos `__hash__()` customizados, fique atento que *hash()* trunca o valor retornado baseado no comprimento de bits da máquina hospedeira. Veja `__hash__()` para mais detalhes.

---

help([*object*])
Invoca o sistema de ajuda embutido. (Esta função é destinada para uso interativo.) Se nenhum argumento é passado, o sistema interativo de ajuda inicia no interpretador do console. Se o argumento é uma string, então a string é pesquisada como o nome de um módulo, função, classe, metódo, palavra-chave, ou tópico de documentação, e a página de ajuda é exibida no console. Se o argumento é qualquer outro tipo de objeto, uma página de ajuda para o objeto é gerada.

Note que se uma barra(/) aparecer na lista de parâmetros de uma função, quando invocando *help()*, significa que os parâmetros anteriores a barra são apenas posicionais. Para mais informações, veja the FAQ entry on positional-only parameters.

Esta função é adicionada ao espaço de nomes embutido pelo módulo *site*.

Alterado na versão 3.4: Mudanças em *pydoc* e *inspect* significam que as assinaturas reportadas para chamáveis agora são mais compreensíveis e consistentes.

hex(*x*)
Converte um número inteiro para uma string hexadecimal em letras minúsculas pré-fixada com "0x". Se *x* não é um objeto *int* do Python, ele tem que definir um método `__index__()` que retorne um inteiro. Alguns exemplos:

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

Se você quer converter um número inteiro para uma string hexadecimal em letras maiúsculas ou minúsculas, com prefixo ou sem, você pode usar qualquer uma das seguintes maneiras:

```
>>> '%#x' % 255, '%x' % 255, '%X' % 255
('0xff', 'ff', 'FF')
```

(continua na próxima página)

(continuação da página anterior)

```
>>> format(255, '#x'), format(255, 'x'), format(255, 'X')
('0xff', 'ff', 'FF')
>>> f'{255:#x}', f'{255:x}', f'{255:X}'
('0xff', 'ff', 'FF')
```

Veja também *format()* para mais informações.

Veja também *int()* para converter uma string hexadecimal para um inteiro usando a base 16.

Nota: Para obter uma string hexadecimal de um ponto flutuante, use o método *float.hex()*.

id(*object*)

Retorna a "identidade" de um objeto. Ele é um inteiro, o qual é garantido que será único e constante para este objeto durante todo o seu ciclo de vida. Dois objetos com ciclos de vida não sobrepostos podem ter o mesmo valor para *id()*.

CPython implementation detail: This is the address of the object in memory.

input([*prompt*])

se o argumento *prompt* estiver presente, escreve na saída padrão sem uma nova linha ao final. A função então lê uma linha da fonte de entrada, converte a mesma para uma string (removendo o caracter de nova linha ao final), e retorna isso. Quando o final do arquivo (EOF / end-of-file) é encontrado, um erro *EOFError* é levantado. Exemplo:

```
>>> s = input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

Se o módulo *readline* foi carregado, então *input()* usará ele para prover edição de linhas elaboradas e funcionalidades de histórico.

class int([*x*])
class int(*x*, *base=10*)

Return an integer object constructed from a number or string *x*, or return `0` if no arguments are given. If *x* defines `__int__()`, `int(x)` returns `x.__int__()`. If *x* defines `__trunc__()`, it returns `x.__trunc__()`. For floating point numbers, this truncates towards zero.

Se *x* não é um número ou se *base* é fornecida, então *x* deve ser uma string, instância de *bytes* ou *bytearray* representando um inteiro literal em base *base*. Opcionalmente, o literal pode ser precedido por + ou − (sem espaço entre eles) e cercado por espaços em branco. Um literal base-n consiste de dígitos de 0 até n-1, com `a` até `z` (ou `A` até `Z`) com valores de 10 até 35. A *base* padrão é 10. Os valores permitidos são 0 e 2–36. Literais em base-2, -8, e -16 podem ser opcionalmente prefixado com `0b`/`0B`, `0o`/`0O`, ou `0x`/`0X`, assim como literais inteiros. Base 0 significa que será interpretado exatamente como um literal, ou seja, as bases são, na verdade, 2, 8, 10, ou 16, e que `int('010', 0)` não é legal, enquanto `int('010')` é, assim como `int('010', 8)`.

O tipo inteiro é descrito em *Tipos Numéricos — int, float, complex*.

Alterado na versão 3.4: Se *base* não é uma instância de *int* e o objeto *base* tem um método `base.__index__`, então esse método é chamado para obter um inteiro para a base. Versões anteriores usavam `base.__int__` ao invés de `base.__index__`.

Alterado na versão 3.6: Agrupar dígitos com sublinhados como em literais de código é permitido.

Alterado na versão 3.7: *x* é agora um parâmetro somente posicional.

Alterado na versão 3.7.14: *int* string inputs and string representations can be limited to help avoid denial of service attacks. A *ValueError* is raised when the limit is exceeded while converting a string *x* to an *int* or

when converting an *int* into a string would exceed the limit. See the *integer string conversion length limitation* documentation.

isinstance(*object*, *classinfo*)
Devolve `True` se o argumento *object* é uma instância argumento *classinfo*, ou de uma subclasse dele (direta, indireta ou *virtual*). Se *object* não é um objeto do tipo dado, a função sempre devolve `False`. Se *classinfo* é uma tupla de tipos de objetos (ou recursivamente, como outras tuplas), devolve `True` se *object* é uma instância de qualquer um dos tipos. Se *classinfo* não é um tipo ou tupla de tipos ou outras tuplas, é lançada uma exceção *TypeError*.

issubclass(*class*, *classinfo*)
Retorna `True` se *class* é uma sub-classe (direta, indireta ou *virtual*) de *classinfo*. Uma classe é considerada uma sub-classe dela mesma. *classinfo* pode ser uma tupla de objetos de classes, e neste caso cada entrada em *classinfo* será verificada. Em qualquer outro caso, uma exceção do tipo *TypeError* é levantada.

iter(*object*[, *sentinel*])
Retorna um objeto *iterator*. O primeiro argumento é interpretado muito diferentemente dependedo da presença do segundo argumento. Sem um segundo argumento, *object* deve ser uma coleção de objetos que suportem o protocolo de iteração (o método `__iter__()`), ou ele deve suportar o protocolo de sequência (o método `__getitem__()` com argumentos inteiros começando em `0`). Se ele não suportar nenhum desses protocolos, um *TypeError* é levantado. Se o segundo argumento, *sentinel*, é fornecido, então *object* deve ser um objeto que pode ser chamado. O iterador criado neste caso irá chamar *object* sem nenhum argumento para cada chamada para o seu métod *__next__()*; se o valor retornado é igual a *sentinel*, então *StopIteration* será levantado, caso contrário o valor será retornado.

Veja também *Tipos de Iteradores*.

Uma aplicação útil da segunda forma de *iter()* é para construir um bloco de leitura. Por exemplo, ler blocos de comprimento fixo de um arquivo binário de banco de dados até que o final do arquivo seja atingido:

```
from functools import partial
with open('mydata.db', 'rb') as f:
    for block in iter(partial(f.read, 64), b''):
        process_block(block)
```

len(*s*)
Retorna o comprimento (o número de itens) de um objeto. O argumento pode ser uma sequência (tal como uma string, bytes, tupla, lista, ou range) ou uma coleção (tal como um dicionário, conjunto, ou conjunto imutável).

class list([*iterable*])
Ao invés de ser uma função, *list* é na verdade uma tipo de sequência mutável, conforme documentado em *Listas* e *Tipos de Sequências — list, tuple, range*.

locals()
Atualiza e retorna um dicionário representando a tabela de símbolos locais atual. Variáveis livres são retornadas por *locals()* quanto ele é chamado em blocos de função, mas não em blocos de classes. Perceba que no nível dos módulos, *locals()* e *globals()* são o mesmo dicionário.

Nota: O conteúdo deste dicionário não deve ser modificado; As alterações podem não afetar os valores das variáveis locais e livres usadas pelo intérprete.

---

map(*function*, *iterable*, ...)
Retorna um iterador que aplica *function* para cada item de *iterable*, devolvendo os resultados. Se argumentos *iterable* adicionais são passados, *function* deve ter a mesma quantidade de argumentos e ela é aplicada aos itens de todos os iteráveis em paralelo. Com múltiplos iteráveis, o iterador para quando o iterador mais curto é encontrado. Para casos onde os parâmetros de entrada da função já estão organizados em tuplas, veja *itertools.starmap()*.

max(*iterable*, *[, *key*, *default*])

max(*arg1*, *arg2*, **args*[, *key*])
Retorna o maior item em um iterável ou o maior de dois ou mais argumentos.

Se um argumento posicional é fornecido, ele deve ser um *iterable*. O maior item no iterável é retornado. Se dois ou mais argumentos posicionais são fornecidos, o maior dos argumentos posicionais é retornado.

Existem dois parâmetros palavra-chave opcionais. O parâmetro *key* especifica uma função de ordenamento de um argumento, como aquelas usadas por `list.sort()`. O parâmetro *default* especifica um objeto a ser retornado se o iterável fornecido estiver vazio. Se o iterável estiver vazion, e *default* não foi fornecido, um `ValueError` é levantado.

Se múltiplos itens são máximos, a função retorna o primeiro encontrado. Isto é consistente com outras ferramentas de ordenamento que preservam a estabilidade, tais como `sorted(iterable, key=keyfunc, reverse=True)[0]` e `heapq.nlargest(1, iterable, key=keyfunc)`.

Novo na versão 3.4: O parâmetro palavra-chave *default*.

class memoryview(*obj*)
Retorna um objeto de "visão da memória" criado a partir do argumento fornecido. Veja *Memory Views* para mais informações.

min(*iterable*, *[, *key*, *default*])
min(*arg1*, *arg2*, **args*[, *key*])
Retorna o menor item de um iterável ou o menor de dois ou mais argumentos.

Se um argumento posicional é fornecido, ele deve ser um *iterable*. O menor item no iterável é retornado. Se dois ou mais argumentos posicionais são fornecidos, o menor dos argumentos posicionais é retornado.

Existem dois parâmetros palavra-chave opcionais. O parâmetro *key* especifica uma função de ordenamento de um argumento, como aquelas usadas por `list.sort()`. O parâmetro *default* especifica um objeto a ser retornado se o iterável fornecido estiver vazio. Se o iterável estiver vazion, e *default* não foi fornecido, um `ValueError` é levantado.

Se múltiplos itens são mínimos, a função retorna o primeiro encontrado. Isto é consistente com outras ferramentas de ordenamento que preservam a estabilidade, tais como `sorted(iterable, key=keyfunc)[0]` e `heapq.nsmallest(1, iterable, key=keyfunc)`.

Novo na versão 3.4: O parâmetro palavra-chave *default*.

next(*iterator*[, *default*])
Recupera o próximo item do *iterator* chamando o seu método `__next__()`. Se *default* foi fornecido, ele é retornado caso o iterável tenha sido percorrido por completo, caso contrário `StopIteration` é levantada.

class object
Retorna um novo objeto sem funcionalidades. `object` é a classe base para todas as classes. Ela tem os métodos que são comuns para todas as instâncias de classes Python classes. Esta função não aceita nenhum argumento.

Nota: `object` *não* tem um atributo `__dict__`, então você não consegue designar atributos arbitrários para uma instância da classe `object`.

---

oct(*x*)
Converte um número inteiro para uma string em base octal, pre-fixada com "0o". O resultado é uma expressão Python válida. Se *x* não for um objeto `int` Python, ele tem que definir um método `__index__()` que retorne um inteiro. Por exemplo:

```
>>> oct(8)
'0o10'
>>> oct(-56)
'-0o70'
```

Se você quiser converter um número inteiro para uma string octal, com o prefixo "0o" ou não, você pode usar qualquer uma das formas a seguir.

```
>>> '%#o' % 10, '%o' % 10
('0o12', '12')
>>> format(10, '#o'), format(10, 'o')
('0o12', '12')
>>> f'{10:#o}', f'{10:o}'
('0o12', '12')
```

Veja também *format()* para mais informações.

open(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)
Abre *file* e retorna um *file object* correspondente. Se o arquivo não puder ser aberto, um an *OSError* é levantado.

file é um *path-like object* fornecendo o caminho (absoluto ou relativo ao diretório de trabalho atual) do arquivo que será aberto, ou de um inteiro descritor de arquivo a ser manipulado (Se um descritor de arquivo é fornecido, ele é fechado quando o objeto de I/O retornado é fechado, a não ser que *closefd* esteja marcado como `False`).

mode é uma string opcional que especifica o modo no qual o arquivo é aberto. O valor padrão é `'r'`, o qual significa abrir para leitura em modo texto. Outros valores comuns são `'w'` para escrever (truncando o arquivo se ele já existe), `'x'` para criação exclusiva e `'a'` para anexar (o qual em *alguns* sistemas Unix, significa que *todas* as escritas anexam ao final do arquivo independentemente da posição de busca atual). No modo texto, se *encoding* não for especificada, a codificação usada é independente de plataforma: `locale.getpreferredencoding(False)` é chamada para obter a codificação da localidade atual (Para ler e escrever bytes diretamente, use o modo binário e não especifique *encoding*). Os modos disponíveis são:

Character	Significado
`'r'`	abre para leitura (padrão)
`'w'`	abre para escrita, truncando o arquivo primeiro (removendo tudo o que estiver contido no mesmo)
`'x'`	abre para criação exclusiva, falhando caso o arquivo exista
`'a'`	abre para escrita, anexando ao final do arquivo caso o mesmo exista
`'b'`	binary mode
`'t'`	modo texto (padrão)
`'+'`	abre um arquivo no disco para atualização (leitura e escrita)

The default mode is `'r'` (open for reading text, synonym of `'rt'`). For binary read-write access, the mode `'w+b'` opens and truncates the file to 0 bytes. `'r+b'` opens the file without truncation.

Conforme mencionado em *Visão Geral*, Python diferencia entre entrada/saída binária e de texto. Arquivos abertos em modo binário (incluindo `'b'` no parâmetro *mode*) retornam o conteúdo como objetos *bytes* sem usar nenhuma decodificação. No modo texto (o padrão, ou quando `'t'` é incluído no parâmetro *mode*), o conteúdo do arquivo é retornado como *str*, sendo os bytes primeiramente decodificados usando uma codificação dependente da plataforma, ou usando a codificação definida em *encoding* se fornecida.

Existe um modo de caracter adicional permitido, `'U'`, o qual não tem mais nenhum efeito, e é considerado como descontinuado. Ele anteriormente habilitava *universal newlines* no modo texto, o que se tornou o comportamento padrão no Python 3.0. Consulte a documentação do parâmetro *newline* para maiores detalhes.

Nota: Python não depende da noção básica do sistema operacional sobre arquivos de texto; todo processamento é feito pelo próprio Python, e é então independente de plataforma.

---

buffering é um inteiro opcional usado para definir a política de buffering. Passe o valor 0 para desativar o buffering (permitido somente em modo binário), passe 1 para selecionar buffering de linha (permitido somente em modo texto), e um inteiro > 1 para indicar o tamanho em bytes de um buffer com tamanho fixo. Quando nenhum valor é fornecido no argumento *buffering*, a política de buffering padrão funciona conforme as seguintes regras:

- Arquivos binários são armazenados em pedaços de tamanho fixo; o tamanho do buffer é escolhido usando uma heurística que tenta determinar o "tamanho de bloco" subjacente do dispositivo, e usa `io.DEFAULT_BUFFER_SIZE` caso não consiga. Em muitos sistemas, o buffer possuirá tipicamente 4096 ou 8192 bytes de comprimento.
- Arquivos de texto "interativos" (arquivos para os quais `isatty()` retornam `True`) usam buffering de linha. Outros arquivos de texto usam a política descrita acima para arquivos binários.

encoding é o nome da codificação usada para codificar ou decodificar o arquivo. Isto deve ser usado apenas no modo texto. A codificação padrão depende da plataforma (seja qual valor `locale.getpreferredencoding()` retornar), mas qualquer *text encoding* suportada pelo Python pode ser usada. Veja o módulo `codecs` para a lista de codificações suportadas.

errors é uma string opcional que especifica como erros de codificação e de decodificação devem ser tratados — isso não pode ser utilizado no modo binário. Uma variedade de tratadores de erro padrão estão disponíveis (listados em *Error Handlers*), apesar que qualquer nome para tratamento de erro registrado com `codecs.register_error()` também é válido. Os nomes padrões incluem:

- `'strict'` para levantar uma exceção `ValueError` se existir um erro de codificação. O valor padrão `None` tem o mesmo efeito.
- `'ignore'` ignora erros. Note que ignorar erros de código pode levar à perda de dados.
- `'replace'` faz um marcador de substituição (tal como `'?'`) ser inserido onde existem dados marformados.
- `'surrogateescape'` representará quaisquer bytes incorretos, conforme códigos apontados na área privada de uso da tabela Unicode, indo desde U+DC80 até U+DCFF. Esses códigos privados serão então convertidos de volta para os mesmos bytes quando o tratamento de erro para `surrogateescape` é usado ao escrever dados. Isto é útil para processar arquivos com uma codificação desconhecida.
- `'xmlcharrefreplace'` é suportado apenas ao gravar em um arquivo. Os caracteres não suportados pela codificação são substituídos pela referência de caracteres XML apropriada `&#nnn;`.
- `'backslashreplace'` substitui dados malformados pela sequência de escape utilizando contrabarra do Python.
- `'namereplace'` (também é suportado somente quando estiver escrevendo) substitui caractere não suportados com sequências de escape `\N{...}`.

newline controla como o modo de *novas linhas universais* funciona (se aplica apenas ao modo texto). Ele pode ser `None`, `''`, `'\n'`, `'\r'` e `'\r\n'`. Ele funciona da seguinte forma:

- Ao ler a entrada do fluxo, se *newline* for `None`, o modo universal de novas linhas será ativado. As linhas na entrada podem terminar em `'\n'`, `'\r'` ou `'\r\n'`, e são traduzidas para `'\n'` antes de retornar ao chamador. Se for `''`, o modo de novas linhas universais será ativado, mas as terminações de linha serão retornadas ao chamador sem tradução. Se houver algum dos outros valores legais, as linhas de entrada são finalizadas apenas pela sequência especificada e a finalização da linha é retornada ao chamador sem tradução.
- Ao gravar a saída no fluxo, se *newline* for `None`, quaisquer caracteres `'\n'` gravados serão traduzidos para o separador de linhas padrão do sistema, `os.linesep`. Se *newline* for `''` ou `'\n'`, nenhuma tradução ocorrerá. Se *newline* for um dos outros valores legais, qualquer caractere `'\n'` escrito será traduzido para a sequência especificada.

Se *closefd* for `False` e um descritor de arquivo em vez de um nome de arquivo for fornecido, o descritor de arquivo subjacente será mantido aberto quando o arquivo for fechado. Se um nome de arquivo for fornecido *closefd* deve ser `True` (o padrão), caso contrário, um erro será levantado.

Um abridor personalizado pode ser usado passando uma chamada como *opener*. O descritor de arquivo subjacente para o objeto de arquivo é obtido chamando *opener* com (*file*, *flags*). *opener* deve retornar um descritor de arquivo aberto (passando *os.open* como *opener* resulta em funcionalidade semelhante à passagem de `None`).

O recém criado arquivo é ref:*non-inheritable 1*.

O exemplo a seguir usa o parâmetro *dir_fd* da função *os.open()* para abrir um arquivo relativo a um determinado diretório:

```
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
...
>>> with open('spamspam.txt', 'w', opener=opener) as f:
...     print('This will be written to somedir/spamspam.txt', file=f)
...
>>> os.close(dir_fd)  # don't leak a file descriptor
```

O tipo de *objeto arquivo* retornado pela função *open()* depende do modo. Quando *open()* é usado para abrir um arquivo no modo texto (`'w'`, `'r'`, `'wt'`, `'rt'`, etc.), retorna uma subclasse de *io.TextIOBase* (especificamente *io.TextIOWrapper*). Quando usada para abrir um arquivo em modo binário com buffer, a classe retornada é uma subclasse de *io.BufferedIOBase*. A classe exata varia: no modo binário de leitura, ele retorna uma *io.BufferedReader*; nos modos binário de gravação e binário anexado, ele retorna um *io.BufferedWriter* e, no modo leitura/gravação, retorna um *io.BufferedRandom*. Quando o buffer está desativado, o fluxo bruto, uma subclasse de *io.RawIOBase*, *io.FileIO*, é retornado.

Veja também os módulos de para lidar com arquivos, tais como, *fileinput*, *io* (onde *open()* é declarado), *os*, *os.path*, *tempfile*, e *shutil*.

Alterado na versão 3.3:

- O parâmetro *opener* foi adicionado.
- O modo `'x'` foi adicionado.
- *IOError* costumava ser levantado, agora ele é um codinome para *OSError*.
- *FileExistsError* agora é levantado se o arquivo aberto no modo de criação exclusivo (`'x'`) já existir.

Alterado na versão 3.4:

- O arquivo agora é não herdável.

Deprecated since version 3.4, will be removed in version 3.9: O modo `'U'`.

Alterado na versão 3.5:

- Se a chamada de sistema é interrompida e o tratador de sinal não levanta uma exceção, a função agora tenta novamente a chamada de sistema em vez de levantar uma exceção *InterruptedError* (consulte **PEP 475** para entender a lógica).
- O manipulador de erro `'namereplace'` foi adicionar.

Alterado na versão 3.6:

- Suporte adicionado para aceitar objetos implementados *os.PathLike*.
- No Windows, a abertura de um buffer do console pode retornar uma subclasse de *io.RawIOBase* que não seja *io.FileIO*.

ord(*c*)
Dada uma sequência que representa um caractere Unicode, retorna um número inteiro representando o ponto de código Unicode desse caractere. Por exemplo, `ord('a')` retorna o número inteiro `97` e `ord('€')` ``(sinal do Euro) retorna ``8364`. Este é o inverso de *chr()*.

pow(*x*, *y*[, *z*])
Return *x* to the power *y*; if *z* is present, return *x* to the power *y*, modulo *z* (computed more efficiently than `pow(x, y) % z`). The two-argument form `pow(x, y)` is equivalent to using the power operator: `x**y`.

The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For *int* operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns `100`, but `10**-2` returns `0.01`. If the second argument is negative, the third argument must be omitted. If *z* is present, *x* and *y* must be of integer types, and *y* must be non-negative.

print(**objects*, *sep=' '*, *end='\n'*, *file=sys.stdout*, *flush=False*)
Imprime *objects* no fluxo de texto *arquivo*, separado por *sep* e seguido por *end*. *sep*, *end*, *file* e *flush*, se houver, devem ser fornecidos como argumentos nomeados.

Todos os argumentos que não são nomeados são convertidos em strings como *str()* faz e gravados no fluxo, separados por *sep* e seguidos por *end*. *sep* e *end* devem ser strings; eles também podem ser `None`, o que significa usar os valores padrão. Se nenhum *object* for fornecido, *print()* escreverá apenas *end*.

O argumento *file* deve ser um objeto com um método `write(string)`; se ele não estiver presente ou `None`, então *sys.stdout* será usado. Como argumentos exibidos no console são convertidos para strings de texto, *print()* não pode ser usado com objetos de arquivo em modo binário. Para esses casos, use `file.write(...)` ao invés.

Se a saída é armazenada em um buffer é usualmente determinado por *file*, mas se o argumento nomeado *flush* é verdadeiro, o fluxo de saída é forçosamente descarregado.

Alterado na versão 3.3: Adicionado o argumento nomeado *flush*.

class property(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)
Retorna um atributo de propriedade.

fget é uma função para obter o valor de um atributo. *fset* é uma função para definir um valor para um atributo. *fdel* é uma função para deletar um valor de um atributo. E *doc* cria um docstring para um atributo.

Um uso comum é para definir um atributo gerenciável `x`:

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

Se *c* é uma instância de *C*, `c.x` irá invocar o método getter, `c.x = value` irá invocar o método setter, e `del c.x` o médodo deleter.

Se fornecido, *doc* será a docstring do atributo property attribute. Otherwise, the property will copy *fget*'s docstring (if it exists). This makes it possible to create read-only properties easily using *property()* as a *decorator*:

```
class Parrot:
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

O decorador `@property` transforma o método `voltage()` em um "getter" para um atributo somente leitura com o mesmo nome, e define a docstring de *voltage* para "Get the current voltage."

Um objeto property possui métodos `getter`, `setter`, e `deleter` usáveis como decoradores, que criam uma cópia da property com o assessor correspondente a função definida para a função com decorador. Isso é explicado melhor com um exemplo:

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

Esse código é exatamente equivalente ao primeiro exemplo. Tenha certeza de nas funções adicionais usar o mesmo nome que a property original (`x` neste caso).

O objeto property retornado também tem os atributos `fget`, `fset`, e `fdel` correspondendo aos argumentos do construtor.

Alterado na versão 3.5: Agora é possível escrever nas docstrings de objetos property.

class range(*stop*)
class range(*start*, *stop*[, *step*])
Em vez de ser uma função, *range* é realmente um tipo de sequência imutável, conforme documentado em *Ranges* e *Tipos de Sequências — list, tuple, range*.

repr(*object*)
Retorna uma string contendo uma representação imprimível de um objeto. Para muitos tipos, essa função tenta retornar uma string que produziria um objeto com o mesmo valor quando passado para *eval()*, caso contrário, a representação é uma string entre colchetes angulares que contém o nome do tipo do objeto juntamente com informações adicionais, geralmente incluindo o nome e o endereço do objeto. Uma classe pode controlar o que essa função retorna para suas instâncias, definindo um método `__repr__()`.

reversed(*seq*)
Retorna um *iterador* reverso. *seq* deve ser um objeto que possui o método `__reversed__()` ou suporta o protocolo de sequência (o método `__len__()` e o método `__len__()` e o método `__getitem__()` com argumentos inteiros começando em `0`).

round(*number*[, *ndigits*])

Retorna *number* arredondado para *ndigits* precisão após o ponto decimal. Se *ndigits* for omitido ou for `None`, ele retornará o número inteiro mais próximo de sua entrada.

Para os tipos embutidos com suporte a *round()*, os valores são arredondados para o múltiplo mais próximo de 10 para a potência de menos *ndigit*; se dois múltiplos são igualmente próximos, o arredondamento é feito para a opção par (por exemplo, `round(0,5)` e `round(-0,5)` são `0` e `round(1.5)` é `2`). Qualquer valor inteiro é válido para *ndigits* (positivo, zero ou negativo). O valor de retorno é um número inteiro se *ndigits* for omitido ou `None`. Caso contrário, o valor de retorno tem o mesmo tipo que *number*.

Para um objeto Python geral `number`, `round` delega para `number.__round__`.

Nota: O comportamento de *round()* para pontos flutuantes pode ser surpreendente: por exemplo, `round(2.675, 2)` fornece `2.67` em vez do esperado `2.68`. Isso não é um bug: é resultado do fato de que a maioria das frações decimais não pode ser representada exatamente como um ponto flutuante. Veja tut-fp-issues para mais informações.

---

class set([*iterable*])
Retorna um novo objeto *set*, opcionalmente com elementos retirados de *iterable*. `set` é uma classe embutida. Veja *set* e *Tipo Set — set, frozenset* para documentação sobre esta classe.

Para outros contêineres, consulte as classes embutidas *frozenset*, *list*, *tuple* e *dict*, bem como o módulo *collections*.

setattr(*object*, *name*, *value*)
Esta é a contrapartida de *getattr()*. Os argumentos são um objeto, uma string e um valor arbitrário. A string pode nomear um atributo existente ou um novo atributo. A função atribui o valor ao atributo, desde que o objeto permita. Por exemplo, `setattr(x, 'foobar', 123)` é equivalente a `x.foobar = 123`.

class slice(*stop*)
class slice(*start*, *stop*[, *step*])
Retorna um objeto *slice* representando o conjunto de índices especificado por `range(start, stop, step)`. Os argumentos *start* e *step* são padronizados como `None`. Os objetos de têm atributos de dados somente leitura `start`, `stop` e `step`, que meramente retornam os valores do argumento (ou o padrão). Eles não têm outra funcionalidade explícita; no entanto, eles são usados pelo Python numérico e outras extensões de terceiros. Os objetos slice também são gerados quando a sintaxe de indexação estendida é usada. Por exemplo: `a[start:stop:step]` ou `a[start:stop, i]`. Veja *itertools.islice()* para uma versão alternativa que retorna um iterador.

sorted(*iterable*, *, *key=None*, *reverse=False*)
Retorna uma nova lista classificada dos itens em *iterable*.

Possui dois argumentos opcionais que devem ser especificados como argumentos nomeados.

key especifica a função de um argumento usado para extrair uma chave de comparação de cada elemento em *iterable* (por exemplo, `key=str.lower`). O valor padrão é `None` (compara os elementos diretamente).

reverse é um valor booleano. Se definido igual a `True`, então os elementos da lista são classificados como se cada comparação fosse reversa.

Usa *functools.cmp_to_key()* para converter a função das antigas *cmp* para uma função *key*.

A função embutida *sorted()* é garantida como estável. Uma ordem é estável se garantir não alterar a ordem relativa dos elementos que se comparam da mesma forma — isso é útil para ordenar em várias passagens (por exemplo, ordenar por departamento e depois por nível de salário).

Para selecionar exemplos e um breve tutorial de classificação, veja: ref: *sortinghowto*.

@staticmethod
Transforma um método em método estático.

Um método estático não recebe um primeiro argumento implícito. Para declarar um método estático, use este idioma:

```
class C:
    @staticmethod
    def f(arg1, arg2, ...): ...
```

A forma `@staticmethod` é uma função de *decorator* – veja function para detalhes.

Um método estático pode ser chamado na classe (tal como `C.f()`) ou em uma instância (tal como `C().f()`).

Métodos estáticos em Python são similares àqueles encontrados em Java ou C++. Veja também *`classmethod()`* para uma variante útil na criação de construtores de classe alternativos.

Como todos os decoradores, também é possível chamar `staticmethod` como uma função regular e fazer algo com seu resultado. Isso é necessário em alguns casos em que você precisa de uma referência para uma função de um corpo de classe e deseja evitar a transformação automática em método de instância. Para esses casos, use este idioma:

```
class C:
    builtin_open = staticmethod(open)
```

Para mais informações sobre métodos estáticos, consulte types.

class str(*object=''*)
class str(*object=b''*, *encoding='utf-8'*, *errors='strict'*)
Retorna uma versão *`str`* de *object*. Consulte *`str()`* para detalhes.

`str` é uma string embutida *class*. Para informações gerais sobre strins, consulte *Tipo de Sequência de Texto — str*.

sum(*iterable*[, *start*])
Sums *start* and the items of an *iterable* from left to right and returns the total. *start* defaults to `0`. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

Para alguns casos de uso, existem boas alternativas para *`sum()`*. A maneira rápida e preferida de concatenar uma sequência de strings é chamando `''.join(sequence)`. Para adicionar valores de ponto flutuante com precisão estendida, consulte *`math.fsum()`*. Para concatenar uma série de iteráveis, considere usar *`itertools.chain()`*.

super([*type*[, *object-or-type*]])
Return a proxy object that delegates method calls to a parent or sibling class of *type*. This is useful for accessing inherited methods that have been overridden in a class. The search order is same as that used by *`getattr()`* except that the *type* itself is skipped.

The *`__mro__`* attribute of the *type* lists the method resolution search order used by both *`getattr()`* and *`super()`*. The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

Se o segundo argumento for omitido, o objeto super retornado é desacoplado. Se o segundo argumento é um objeto, `isinstance(obj, type)` deve ser verdadeiro. Se o segundo argumento é um tipo, `issubclass(type2, type)` deve ser verdadeiro (isto é útil para classmethods).

Existem dois casos de uso típicos para *super*. Em uma hierarquia de classes com herança única, *super* pode ser usado para se referir a classes-pai sem nomeá-las explicitamente, tornando o código mais sustentável. Esse uso é paralelo ao uso de *super* em outras linguagens de programação.

O segundo caso de uso é oferecer suporte à herança múltipla cooperativa em um ambiente de execução dinâmica. Esse caso de uso é exclusivo do Python e não é encontrado em idiomas ou linguagens compiladas estaticamente que suportam apenas herança única. Isso torna possível implementar "diagramas em losango", onde várias classes base implementam o mesmo método. Um bom design determina que esse método tenha a mesma assinatura de chamada em todos os casos (porque a ordem das chamadas é determinada em tempo de execução, porque essa ordem se

adapta às alterações na hierarquia de classes e porque essa ordem pode incluir classes de irmãos desconhecidas antes do tempo de execução).

Nos dois casos de uso, uma chamada típica de superclasse se parece com isso:

```
class C(B):
    def method(self, arg):
        super().method(arg)     # This does the same thing as:
                                # super(C, self).method(arg)
```

In addition to method lookups, `super()` also works for attribute lookups. One possible use case for this is calling *descriptor*s in a parent or sibling class.

Observe que `super()` é implementada como parte do processo de vinculação para procura explícita de atributos com ponto, tal como `super().__getitem__(nome)`. Ela faz isso implementando seu próprio método `__getattribute__()` para pesquisar classes em uma ordem predizível que possui suporte a herança múltipla cooperativa. Logo, `super()` não é definida para procuras implícitas usando instruções ou operadores como `super()[name]`.

Observe também que, além da forma de argumento zero, `super()` não se limita ao uso de métodos internos. O formulário de dois argumentos especifica exatamente os argumentos e faz as referências apropriadas. O formulário de argumento zero funciona apenas dentro de uma definição de classe, pois o compilador preenche os detalhes necessários para recuperar corretamente a classe que está sendo definida, além de acessar a instância atual para métodos comuns.

Para sugestões práticas sobre como projetar classes cooperativas usando `super()`, consulte o guia para uso de super().

class tuple([*iterable*])
Ao invés de ser uma função, `tuple` é na verdade um tipo de sequência imutável, conforme documentado em *Tuplas* e *Tipos de Sequências — list, tuple, range*.

class type(*object*)
class type(*name*, *bases*, *dict*)
Com um argumento, retorna o tipo de um *object*. O valor de retorno é um tipo de objeto e geralmente o mesmo objeto retornado por `object.__class__`.

A função embutida `isinstance()` é recomendada para testar o tipo de um objeto, porque ela leva sub-classes em consideração.

Com três argumentos, retorna um novo objeto type. Esta é essencialmente a forma dinâmica da instrução `class`. A string *name* é o nome da classe e se torna o atributo `__name__`; a tupla *bases* transforma em itens as classes bases e se torna o atributo `__bases__`; e o dicionário *dict* é o namespace contendo as definições para a classe principal, e é copiada para um dicionário padrão para se tornar o atributo `__dict__`. Por exemplo, as seguintes duas instruções criam objetos `type` objects: idênticos

```
>>> class X:
...     a = 1
...
>>> X = type('X', (object,), dict(a=1))
```

Veja também *Objetos de tipo*.

Alterado na versão 3.6: Sub-classes de `type` que não fazem sobrecarga de `type.__new__` não podem mais usar a forma com apenas um argumento para obter o tipo de um objeto.

vars([*object*])
Retorna o atributo `__dict__` para um módulo, classe, instância, or qualquer outro objeto com um atributo `__dict__`.

Objetos como modelos e instâncias têm um atributo atualizável *__dict__*; porém, outros projetos podem ter restrições de escrita em seus atributos *__dict__* (por exemplo, classes usam um *types.MappingProxyType* para prevenir atualizações diretas a dicionário).

Sem um argumento, *vars()* funciona como *locals()*. Perceba que, o dicionário locals é apenas útil para leitura, pelo fato de alterações no dicionário locals serem ignoradas.

zip(**iterables*)

Produzi um iterador que agrega elementos de cada um dos iteráveis.

Retorna um iterador de tuplas, onde a *i*ª tupla contém o *i*º elemento de cada uma das sequências de argumentos ou iteráveis. O iterador é parado quando a menor entrada iterável é esgotada. Com um único argumento iterável, ele retorna um iterador de 1 tupla. Sem argumentos, ele retorna um iterador vazio. Equivalente a:

```
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

A ordem de avaliação da esquerda para a direita dos iteráveis é garantida. Isso possibilita um idioma para agrupar uma série de dados em grupos de comprimento n usando `zip(*[iter(s)]*n)`. Isso repete o *mesmo* iterador `n` vezes, para que cada tupla de saída tenha o resultado de `n` chamadas ao iterador. Isso tem o efeito de dividir a entrada em pedaços de comprimento n.

zip() deve ser usado apenas com entradas de comprimento diferente quando você não se importa com valores inigualáveis à direita de iteráveis mais longos. Se esses valores forem importantes, use *itertools.zip_longest()*.

zip() em conjunto com o operador `*` pode ser usado para descompactar uma lista:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped)
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zip(x, y))
>>> x == list(x2) and y == list(y2)
True
```

__import__(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)

Nota: Esta é uma função avançada que não é necessária na programação diária do Python, ao contrário de *importlib.import_module()*.

Esta função é chamada pela instrução `import`. Ela pode ser substituída (importando o módulo *builtins* e atribuindo a `builtins.__import__`) para alterar a semântica da instrução `import`, mas isso é **fortemente** desencorajado, pois geralmente é mais simples usar ganchos de importação (consulte: pep: *302*) para atingir os mesmos objetivos e não causa problemas com o código que pressupõe que a implementação de importação

padrão esteja em uso. O uso direto de *__import__()* também é desencorajado em favor de *importlib.import_module()*.

A função importa o módulo *name*, potencialmente usando os dados *globals* e *locals* para determinar como interpretar o nome em um contexto de pacote. O *fromlist* fornece os nomes de objetos ou submódulos que devem ser importados do módulo, fornecidos por *name*. A implementação padrão não usa seu argumento *locals* e usa seus *globals* apenas para determinar o contexto do pacote da instrução `import`.

level especifica se é necessário usar importações absolutas ou relativas. `0` (o padrão) significa apenas realizar importações absolutas. Valores positivos para *level* indicam o número de diretórios pai a serem pesquisados em relação ao diretório do módulo que chama *__import__()* (consulte **PEP 328** para obter detalhes).

Quando a variável *name* está no formato `package.module`, normalmente, o pacote de nível superior (o nome até o primeiro ponto) é retornado, *não* o módulo nomeado por *name*. No entanto, quando um argumento *fromlist* não vazio é fornecido, o módulo nomeado por *name* é retornado.

Por exemplo, a instrução `importar spam` resulta em bytecode semelhante ao seguinte código:

```
spam = __import__('spam', globals(), locals(), [], 0)
```

A instrução `import spam.ham` resulta nesta chamada:

```
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

Observe como *__import__()* retorna o módulo de nível superior aqui, porque este é o objeto vinculado a um nome pela instrução `import`.

Por outro lado, a instrução "from spam.ham import eggs, sausage as saus" resulta em

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], 0)
eggs = _temp.eggs
saus = _temp.sausage
```

Aqui, o módulo `spam.ham` é retornado de *__import__()*. A partir desse objeto, os nomes a serem importados são recuperados e atribuídos aos seus respectivos nomes.

Se você simplesmente deseja importar um módulo (potencialmente dentro de um pacote) pelo nome, use *importlib.import_module()*.

Alterado na versão 3.3: Valores negativos para *level* não são mais suportados (o que também altera o valor padrão para 0).

CAPÍTULO 3

Constantes Built-in

Um pequeno número de constantes são definidas no espaço de nomes embutido da linguagem. São elas:

`False`
O valor falso do tipo *`bool`*. As atribuições a `False` são ilegais e levantam *`SyntaxError`*.

`True`
O valor verdadeiro do tipo *`bool`*. As atribuições a `True` são ilegais e levantam *`SyntaxError`*.

`None`
The sole value of the type `NoneType`. `None` is frequently used to represent the absence of a value, as when default arguments are not passed to a function. Assignments to `None` are illegal and raise a *`SyntaxError`*.

`NotImplemented`
Special value which should be returned by the binary special methods (e.g. `__eq__()`, `__lt__()`, `__add__()`, `__rsub__()`, etc.) to indicate that the operation is not implemented with respect to the other type; may be returned by the in-place binary special methods (e.g. `__imul__()`, `__iand__()`, etc.) for the same purpose. Its truth value is true.

Nota: Quando um método binário (ou local) retorna `NotImplemented`, o interpretador tentará a operação refletida no outro tipo (ou algum outro fallback, dependendo do operador). Se todas as tentativas retornarem `NotImplemented`, o interpretador levantará uma exceção apropriada. Retornar incorretamente `NotImplemented` resultará em uma mensagem de erro enganosa ou no valor `NotImplemented` sendo retornado ao código Python.

Consulte *Implementando as operações aritméticas* para ver exemplos.

Nota: `NotImplementedError` e `NotImplemented` não são intercambiáveis, mesmo que tenham nomes e propósitos similares. Veja *`NotImplementedError`* para detalhes e casos de uso.

`Ellipsis`
The same as the ellipsis literal "`...`". Special value used mostly in conjunction with extended slicing syntax for user-defined container data types.

__debug__
Esta constante é verdadeira se o Python não foi iniciado com uma opção `-O`. Veja também a instrução `assert`.

Nota: Os nomes *None*, *False*, *True* e *__debug__* não podem ser reatribuídos (atribuições a eles, mesmo como um nome de atributo, levantam *SyntaxError*), para que possam ser consideradas "verdadeiras" constantes.

3.1 Constantes adicionadas pelo módulo `site`

O módulo *site* (que é importado automaticamente durante a inicialização, exceto se a opção de linha de comando `-S` for fornecida) adiciona várias constantes ao espaço de nomes embutido. Eles são úteis para o console do interpretador interativo e não devem ser usados em programas.

quit(*code=None*)
exit(*code=None*)
Objetos que, quando impressos, imprimem uma mensagem como "Use quit() or Ctrl-D (i.e. EOF) to exit" e, quando chamados, levantam *SystemExit* com o código de saída especificado.

copyright
credits
Objetos que ao serem impressos ou chamados, imprimem o texto dos direitos autorais ou créditos, respectivamente.

license
Objeto que, quando impresso, imprime a mensagem "Type license() to see the full license text" e, quando chamado, exibe o texto completo da licença de maneira semelhante a um paginador (uma tela por vez).

CAPÍTULO 4

Tipos internos

As seções a seguir descrevem os tipos padrão que são incorporados ao interpretador.

Os principais tipos internos são numéricos, sequências, mapeamentos, classes, instâncias e exceções.

Algumas classes coletadas são mutáveis. Os métodos que adicionam, subtraem, ou reorganizam seus membros no lugar, e não retornam um item específico, nunca retornam a instância da coleção propriamente dita, mas um `None`.

Alguns operadores são suportados por diversos tipos de objeto; em particular, praticamente todos os objetos podem ser comparados, testados para o valor verdade, e convertidos para uma string (com a função *`repr()`* ou a função ligeiramente diferente *`str()`*). A última função nesse caso é implicitamente usada quando um objeto é escrito pela função *`print()`*.

4.1 Teste do Valor Verdade

Qualquer objeto pode ser testado quanto ao valor de verdade, para uso em uma condição `if` ou `while` ou como operando das operações booleanas abaixo.

Por padrão, um objeto é considerado verdadeiro, a menos que sua a classe defina um método `__bool__()` que retorne `False` ou um método método :meth:‘ __len__‘ que retorna zero, quando chamado com o objeto.[1] Aqui estão a maioria dos objetos built-ins considerados falsos:

- constantes definidas como False: `None` e `False`.
- zero de qualquer tipo numérico: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- sequências vazias e coleções: `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

Operações e funções embutidas que têm um resultado Booleano retornam `0` ou `False` para falso e `1` ou `True` para verdadeiro, salvo indicações ao contrário. (Exceção importante: as operações Booleanas `or` e `and` sempre retornam um de seus operandos.)

[1] Informações adicionais sobre esses métodos especiais podem ser encontradas no Manual de Referência do Python (Customização básica).

4.2 Operações booleanas — `and`, `or`, `not`

Esses são as operações Booleanas, ordenados por prioridade ascendente:

Operação	Resultado	Notas
`x or y`	se *x* é falso, então *y*, do contrário *x*	(1)
`x and y`	se *x* é falso, então *x*, do contrário *y*	(2)
`not x`	se *x* é falso, então `True`, caso contrário `False`	(3)

Notas:

(1) Esse é um operador de curto-circuito, por isso só avalia o segundo argumento se o primeiro é falso.

(2) Este é um operador de curto-circuito, por isso só avalia o segundo argumento se o primeiro é verdadeiro.

(3) `not` tem uma baixa prioridade do que operadores não-Booleanos, então `not a == b` é interpretado como `not (a == b)`, e `a == not b` é um erro de sintaxe.

4.3 Comparações

Há oito operadores comparativos no Python. Todos eles possuem a mesma prioridade (que é maior do que aquela das operações Booleanas). Comparações podem ser acorrentadas arbitrariamente; por exemplo, `x < y <= z` é equivalente a `x < y and y <= z`, exceto que *y* é avaliado apenas uma vez (porém em ambos os casos *z* não é avaliado de todo quando `x < y` é sabido ser falso).

Esta tabela resume as operações de comparação:

Operação	Significado
`<`	estritamente menor que
`<=`	menor que ou igual
`>`	estritamente maior que
`>=`	maior que ou igual
`==`	igual
`!=`	não é igual
`is`	identidade do objeto
`is not`	identidade de objeto negada

Objetos de tipos diferentes, exceto tipos diferentes numéricos, nunca comparam igual. Além disso, alguns tipos (por exemplo, objetos função) suportam apenas uma noção degenerada de comparação onde dois objetos desse tipo são desiguais. O operador <, <=, > e >= irá elencar a exceção *`TypeError`* quando comparando um número complexo com outro tipo numérico embutido, quando os objetos são de diferentes tipos que não podem ser comparados, ou em outros casos quando não há ordem definida.

Instâncias não idênticas de uma classe normalmente comparam-se como desiguais ao menos que a classe defina o método `__eq__()`.

Instâncias de uma classe não podem ser ordenadas com respeito a outras instâncias da mesma classe, ou outros tipos de objeto, ao menos que a classe defina o suficiente de métodos `__lt__()`, `__le__()`, `__gt__()`, e `__ge__()` (no geral, `__lt__()` e `__eq__()` são suficientes, se você quiser o significado convencional dos operadores de comparação).

O comportamento dos operadores de palavra chave `is` e `is not` não podem ser personalizados; além disso eles podem ser aplicados a qualquer objeto e nunca criam uma exceção.

Mais duas operações com a mesma prioridade sintática, `in` e `not in`, são suportadas por tipos que são *iterable* ou implementam o método `__contains__()`.

4.4 Tipos Numéricos — `int`, `float`, `complex`

Há três tipos numéricos distintos: *integers*, *floating point numbers*, e *complex numbers*. Além do mais, Booleanos são um subtipo dos integrais. Integrais possuem precisão ilimitada. Números de ponto flutuante são usualmente implementados usando `double` em C; informação sobre a precisão e representação interna de números com ponto flutuante para a máquina na qual o seu programa está rodando é disponibilizada em `sys.float_info`. Números complexos possuem uma parte real e imaginária, no qual cada uma é um número de ponto flutuante. Para extrair essas partes de um número complexo *z*, use `z.real` e `z.imag`. (A biblioteca padrão inclui tipos numéricos adicionais, `fractions` que possuem racionais, e `decimal` que possuem números de ponto flutuante com precisão definida pelo usuário).

Números são criados por literais numéricos ou como resultado de funções embarcadas e operadoras. Integrais literais planos (incluindo números hexadecimais, octais e binários) culminam em integrais. Literais numéricos contendo um ponto decimal ou um sinal exponencial resultam em números de ponto flutuante. Anexando `'j'` ou “'J'“para um literal numérico resulta em um número imaginário (um número complexo com uma parte real zero) com a qual você pode adicionar a um integral ou flutuante para receber um número complexo com partes reais e imaginárias.

Python suporta completamente aritmética mista: quando um operador de aritmética binária tem operandos de tipos numéricos diferentes, o operando com o tipo mais “estreito” é ampliado para o tipo do outro operando, onde um inteiro é mais estreito do que um ponto flutuante, que por sua vez é mais estreito que um número complexo. Uma comparação entre números de diferentes tipos se comporta como se os valores exatos desses números estivessem sendo comparados.[2]

Os construtores `int()`, `float()`, e `complex()` podem ser usados para produzir números de um tipo específico.

Todos os tipos numéricos (exceto complexos) suportam as seguintes operações (para prioridades das operações, consulte operator-summary):

Operação	Resultado	Notas	Documentação completa
`x + y`	soma de *x* e *y*		
`x - y`	diferença de * x * e * y *		
`x * y`	produto de * x * e * y *		
`x / y`	quociente de *x* e *y*		
`x // y`	quociente flutuante of *x* and *y*	(1)	
`x % y`	restante de `x / y`	(2)	
`-x`	*x* negado		
`+x`	*x* inalterado		
`abs(x)`	valor absoluto ou magnitude de *x*		`abs()`
`int(x)`	*x* convertido em inteiro	(3)(6)	`int()`
`float(x)`	*x* convertido em ponto flutuante	(4)(6)	`float()`
`complex(re, im)`	um número complexo com parte real *re*, parte imaginária *im*. *im* acarreta em zero.	(6)	`complex()`
`c.conjugate()`	conjugado do número complexo *c*		
`divmod(x, y)`	o par `(x // y, x % y)`	(2)	`divmod()`
`pow(x, y)`	*x* elevado a *y*	(5)	`pow()`
`x ** y`	*x* elevado a *y*	(5)	

Notas:

[2] Como uma consequência, a lista `[1, 2]` é considerada igual a `[1.0, 2.0]`, e similarmente para tuplas.

(1) Também referido como uma divisão inteira. O valor resultante é um integral inteiro, embora o tipo oriundo do resultado não seja necessariamente int. O resultado é sempre arredondado para menos infinito: `1//2` é `0`, `(-1)//2` é `-1`, `1//(-2)` é `-1`, e `(-1)//(-2)` é `0`.

(2) Não para números complexos. Ao invés disso converte para flutuantes usando *abs()* se for apropriado.

(3) Conversão de ponto flutuante para inteiro pode arredondar ou truncar como ocorre em C; veja as funções *math.floor()* e *math.ceil()* para conversões bem definidas.

(4) ponto flutuante também aceita a string "nan" e "inf" com um prefixo opcional "+" ou "-" para Não é um Número (NaN) e infinidade positiva ou negativa.

(5) Python define `pow(0, 0)` e `0 ** 0` sendo `1`, como é comum para linguagens de programação.

(6) Os literais numéricos aceitados incluem os dígitos de `0` a `9` ou qualquer equivalente Unicode (pontos de código com a propriedade `Nd`).

See http://www.unicode.org/Public/10.0.0/ucd/extracted/DerivedNumericType.txt for a complete list of code points with the `Nd` property.

Todos os tipos *numbers.Real* (*int* and *float*) também incluem as seguintes operações.

Operação	Resultado
math.trunc(x)	*x* truncado para *Integral*
:func:‘round(x[, n]) 1‘<round>	*x* arredondado para *n* dígitos, arredondando metade para igualar. Se *n* é omitido, ele toma o padrão de 0.
math.floor(x)	o maior *Integral* <= *x*
math.ceil(x)	pelo menos *Integral* >= *x*

Para operações numéricas adicionais, consulte os módulos *math* e *cmath*.

4.4.1 Operações de bits em tipos inteiros

Operações bit a bit só fazem sentido para números inteiros. O resultado de operações bit a bit é calculado como se fosse realizado no complemento de dois com um número infinito de bits de sinal.

As prioridades das operações bitwise binárias são todas menores do que as operações numéricas e maiores que as comparações; a operação unária ~ tem a mesma prioridade que as outras operações numéricas unárias (+ e −).

Esta tabela lista as operações de bits classificadas em prioridade ascendente:

Operação	Resultado	Notas
`x | y`	bitwise *or* de *x* e *y*	(4)
`x ^ y`	bitwise *exclusive or* de *x* e *y*	(4)
`x & y`	bitwise *and* de *x* e *y*	(4)
`x << n`	*x* deslocado para a esquerda pelos bits *n*	(1)(2)
`x >> n`	*x* deslocado para a direita pelos bits *n*	(1)(3)
`~x`	os bits de *x* invertidos	

Notas:

(1) Contagens de deslocamento negativo são ilegais e causam o acionamento de um *ValueError*.

(2) A left shift by *n* bits is equivalent to multiplication by `pow(2, n)`.

(3) A right shift by *n* bits is equivalent to floor division by `pow(2, n)`.

(4) Executar esses cálculos com pelo menos um bit de extensão de sinal extra na representação de complemento de dois finitos (uma largura de bit de trabalho `1+max(x.bit_length(), y.bit_length())` ou mais) é suficiente para obter o mesmo resultado como se houvesse um número infinito de bits de sinal.

4.4.2 Métodos adicionais em tipos inteiros

O tipo int implementa a classe *`numbers.Integral` abstract base class*. Além disso, ele provê mais alguns métodos:

`int.`**`bit_length`**`()`
Retornar o número de bits necessários para representar um inteiro em binário, excluindo o sinal e entrelinha zeros:

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

Mais precisamente, se `x` for diferente de zero, então `x.bit_length()` é o único integral positivo `k` tal que“2**(k-1) <= abs(x) < 2**k“. Equvalentemente, quando `abs(x)` for menor o suficiente para ter um arredondamento algorítmicamente correto, então `k = 1 + int(log(abs(x), 2))`. Se `x` é zero, então `x.bit_length()` retorna `0`.

Equivalente a:

```
def bit_length(self):
    s = bin(self)       # binary representation:  bin(-37) --> '-0b100101'
    s = s.lstrip('-0b') # remove leading zeros and minus sign
    return len(s)       # len('100101') --> 6
```

Novo na versão 3.1.

`int.`**`to_bytes`**`(`*length*, *byteorder*, *, *signed=False*`)`
Retorna um array de bytes representando um inteiro.

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() + 7) // 8, byteorder='little')
b'\xe8\x03'
```

O inteiro é representado usando *length* bytes. Um *`OverflowError`* é elencado se um inteiro não é representável com o dado número de bytes.

O argumento *byteorder* determina a ordem de bytes usada para representar um inteiro. Se o *byteorder* é `"big"`, o byte mais significante está no início do array de byte. Se *byteorder* é `"little"`, o byte mais significante está no final do array de byte. Para requisitar a ordem nativa de byte do sistema hospedeiro, use *`sys.byteorder`* como o valor da ordem de byte.

O argumento *signed* determina aonde o modo de complemento de dois é usado para representar o inteiro. Se *signed* é `False` e um inteiro negativo é dado, um *`OverflowError`* é disparado. O valor padrão para *signed* é `False`.

Novo na versão 3.2.

`classmethod` `int.`**`from_bytes`**`(`*bytes*, *byteorder*, *, *signed=False*`)`
Retorna o inteiro representado pelo dado array de bytes.

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

O argumento *bytes* precisa ou ser um *bytes-like object* ou um iterador produzindo bytes.

O argumento *byteorder* determina a ordem de bytes usada para representar um inteiro. Se o *byteorder* é `"big"`, o byte mais significante está no início do array de byte. Se *byteorder* é `"little"`, o byte mais significante está no final do array de byte. Para requisitar a ordem nativa de byte do sistema hospedeiro, use *`sys.byteorder`* como o valor da ordem de byte.

O arguemento *signed* indica quando o complemento de dois é usado para representar o inteiro.

Novo na versão 3.2.

4.4.3 Métodos Adicionais em Ponto Flutuante

O tipo float implementa *`numbers.Real` abstract base class*. float também possui os seguintes métodos adicionais.

`float.`**`as_integer_ratio`**`()`
Retorna um par de inteiros dos quais a proporção é exatamente igual ao float original e com um denominador positivo. Levanta um *`OverflowError`* em infinidades e um *`ValueError`* em NaNs.

`float.`**`is_integer`**`()`
Retornar `True` se a instância do float for finita com o valor integral e `False`, caso contrário:

```
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

Dois métodos suportam conversão para e de cadeias hexadecimais. Uma vez que os flutuadores do Python são armazenados internamente como números binários, a conversão de um flutuador para ou de uma seqüência *decimal* geralmente envolve um pequeno erro de arredondamento. Em contraste, as frases hexadecimais permitem a representação exata e a especificação de números de ponto flutuante. Isso pode ser útil na depuração e no trabalho numérico.

`float.`**`hex`**`()`
Retorna a representação de um número de ponto-flutuante como uma string hexadecimal. Para números de ponto-flutuante finitos, essa representação vai sempre incluir um `0x` inicial e um `p` final e expoente.

`classmethod` `float.`**`fromhex`**`(`*s*`)`
Método de classe para retornar um float representado por uma string hexadecimal *s*. A string *s* pode ter espaços em branco iniciais e finais.

Note que *`float.hex()`* é um método de instância, enquanto *`float.fromhex()`* é um método de classe.

Uma string hexadecimal toma a forma:

```
[sign] ['0x'] integer ['.' fraction] ['p' exponent]
```

aonde o sinal `sign` opcional pode ser tanto `+` or `-`, `integer` e `fraction` são strings de dígitos hexadecimais, e `exponent` é um inteiro decimal com um símbolo precedente opcional. Case não é significante, e deve haver ao menos

um dígito hexadecimal tanto no inteiro ou na fração. Essa síntaxe é similar à síntaxe especificada na seção 6.4.4.2 do padrão C99, e também do da síntaxe usada no Java 1.5 em diante. Em particular, a saída de *float.hex()* é usável como um literal hexadecimal de ponto-flutuante em código C ou Java, e hexadecimal strings produzidas pelo formato do caráctere do C's `%a``ou ``Double.toHexString` do Java são aceitos pelo *float.fromhex()*.

Note que o expoente é escrito em decimal ao invés de hexadecimal, e que ele dá a potência de 2 pela qual se multiplica o coeficiente. Por exemplo, a string hexadecimal `0x3.a7p10` representa o número de ponto-flutuante `(3 + 10./16 + 7./16**2) * 2.0**10`, ou `3740.0`:

```
>>> float.fromhex('0x3.a7p10')
3740.0
```

Aplicando a conversão reversa a `3740.0` retorna uma string hexadecimal diferente representada pelo mesmo número:

```
>>> float.hex(3740.0)
'0x1.d380000000000p+11'
```

4.4.4 Hashing de tipos numéricos

Para números `x` e `y`, possivelmente de diferentes tipos, é um requerimento que `hash(x) == hash(y)` sempre que `x == y` (veja o método: meth:*__hash__* e sua documentação para mais detalhes). Para facilitar a implementação e eficiência através de uma varidade de tipos numéricos (incluindo: class:*int*, *float*, *decimal.Decimal* e *fractions.Fraction*), o hash do Python para tipos numéricos é baseado em uma única funcão matemática que é definida para qualquer número racional, e portanto se aplica para todas as instâncias de *int* e *fractions.Fraction*, e todas as instâncias finitas das classes *float* e *decimal.Decimal*. Essencialmente, essa função é dada pelo modulo de redução `P` para um primo fixado `P`. O valor de `P` é disponibilizado ao Python como um atributo `modulus` do *sys.hash_info*.

CPython implementation detail: Atualmente, o primo usado é `P = 2**31 - 1` em máquinas com 32-bit e inteiros C longos e `P = 2**61 - 1` em máquinas com longs C de 64-bit.

Aqui estão as regras em detalhe:

- Se `x = m / n` é um número racional não negativo e `n` não é divisível por `P`, defina `hash(x)` como `m * invmod(n, P) % P`, aonde `invmod(n, P)` retorna o inverso de `n` modulo `P`.
- Se `x = m / n` é um número racional não negativo e `n` é divisível por `P` (porém `m` não é) então `n` não possui modulo `P` inverso e a regra acima não se aplica; nesse caso defina `hash(x)` para ser o valor constante `sys.hash_info.inf`.
- Se `x = m / n` é um número racional negativo, defina `hash(x)` como `-hash(-x)`. Se a hash resultante é `-1`, substitua ela com `-2`.
- Os valores particulares `sys.hash_info.inf`, `-sys.hash_info.inf` e `sys.hash_info.nan` são usados como valores de hash para infinidade positiva, infinidade negativa, ou nans (respectivamente). (Todos os nans "hasheáveis" possuem o mesmo valor de hash.)
- Para *complex* número `z`, o valor da hash do número real partes imaginárias são combinados computando `hash(z.real) + sys.hash_info.imag * hash(z.imag)`, modulo reduzido `2**sys.hash_info.width` de modo que isto permaneça em `range(-2**(sys.hash_info.width - 1), 2**(sys.hash_info.width - 1))`. Novamente, se os resultados são `-1`, eles são substituídos com `-2`.

Para clarificar as regras acima, aqui estão alguns exemplos de código em Python, equivalentes às hash's embutidas, para computar a hash de números racionais, *float*, ou *complex*:

```
import sys, math
```

(continua na próxima página)

(continuação da página anterior)

```
def hash_fraction(m, n):
    """Compute the hash of a rational number m / n.

    Assumes m and n are integers, with n positive.
    Equivalent to hash(fractions.Fraction(m, n)).

    """
    P = sys.hash_info.modulus
    # Remove common factors of P.  (Unnecessary if m and n already coprime.)
    while m % P == n % P == 0:
        m, n = m // P, n // P

    if n % P == 0:
        hash_value = sys.hash_info.inf
    else:
        # Fermat's Little Theorem: pow(n, P-1, P) is 1, so
        # pow(n, P-2, P) gives the inverse of n modulo P.
        hash_value = (abs(m) % P) * pow(n, P - 2, P) % P
    if m < 0:
        hash_value = -hash_value
    if hash_value == -1:
        hash_value = -2
    return hash_value

def hash_float(x):
    """Compute the hash of a float x."""

    if math.isnan(x):
        return sys.hash_info.nan
    elif math.isinf(x):
        return sys.hash_info.inf if x > 0 else -sys.hash_info.inf
    else:
        return hash_fraction(*x.as_integer_ratio())

def hash_complex(z):
    """Compute the hash of a complex number z."""

    hash_value = hash_float(z.real) + sys.hash_info.imag * hash_float(z.imag)
    # do a signed reduction modulo 2**sys.hash_info.width
    M = 2**(sys.hash_info.width - 1)
    hash_value = (hash_value & (M - 1)) - (hash_value & M)
    if hash_value == -1:
        hash_value = -2
    return hash_value
```

4.5 Tipos de Iteradores

Python suporta o conceito de iteração sobre conteineres. Isso é implementado usando dois métodos distintos; estes são usados para permitir classes definidas pelo usuário suportem iteração. Sequências, descritas abaixo em mais detalhes, sempre suportam os métodos de iteração.

Um método necessita ser definido para objetos conteineres afim destes proverem suporte a iteração:

`container.__iter__()`
Retorna um objeto iterador. O objeto é requerido suportar o protocolo iterador descrito abaixo. Se um conteiner

suporta diferentes tipos de iterador, métodos adicionais podem ser providenciados para requisitar especificamente iteradores para aqueles tipos de iterações. (Um exemplo de um object suportando múltiplas formas de iteração seria uma estrutura em árvore a qual suporta ambas travessias de breadth-first e depth-first.) Esse method corresponde ao slot `tp_iter` do type de estrutura para objetos em Python na API Python/C.

Os objetos iterator por eles mesmos são requeridos que suportem os dois seguintes métodos, que juntos formam o *iterator protocol*:

`iterator.__iter__()`
Retorna o próprio iterator object. Isso é necessário para permitir que ambos os conteineres e iteradores sejam usados com as declarações `for` e `in`. Esse method corresponde ao slot `tp_iter` da estrutura type para objetos do Python na API Python/C.

`iterator.__next__()`
Retorna o próximo item do conteiner. Se não houver itens além, a exceção *`StopIteration`* é levantada. Esse method corresponde ao slot `tp_iternext` do type de estrutura para objetos Python na API Python/C.

Python define diversos objetos iterator para suportar iterações sobre tipos de sequências gerais e específicas, dicionários, e outras formas mais especializadas. Os tipos específicos não são importantes além de sua implementação do protocolo iterator.

Uma vez que o método iterador *`__next__()`* levantou *`StopIteration`*, ele deve continuar fazendo isso em chamadas subsequentes. Implementações que não obedecem essa propriedade são consideradas quebradas.

4.5.1 Tipos de Geradores

Python's *generator*s provê uma maneira conveniente para implementar o protocolo iterator. Se um object conteiner de method `__iter__()` é implementado como um gerador, ele irá automaticamente retornar um object iterator (tecnicamente, um generator object) suprindo os métodos `__iter__()` e `__next__()` . Mais informações sobre geradores podem ser encontradas em a documentação para a expressão yield.

4.6 Tipos de Sequências — `list`, `tuple`, `range`

Existem três tipos básicos de seqüência: listas, tuplas e objetos range. Tipos de sequência adicionais adaptados para o processamento de *binary data* e *text strings* são descritos em seções dedicadas.

4.6.1 Operações de Sequências Comuns

As operações nas seguintes tabelas são suportadas pela maioria das sequências de tipos, ambos mutáveis e imutáveis. A classe *`collections.abc.Sequence`* ABC é fornecida para tornar fácil a correta implementação desses operadores em sequências de tipos customizáveis.

Essa tabela lista as operações de sequência listadas em prioridade ascendente. Na tabela, *s* e *t* são sequências do mesmo type, *n*, *i*, *j* e *k* são inteiros e *x* é um object arbitrário que atende a qualquer restrição de valor e type impostas por *s*.

As operações `in` e `not in` têm as mesmas prioridades que as operações de comparação. As operações + (concatenação) e `*` (repetição) têm a mesma prioridade que as operações numéricas correspondentes.[3]

[3] Eles precisam ter, já que o analisador sintático não consegue dizer o tipo dos operandos.

Operação	Resultado	Notas
`x in s`	`True` caso um item de *s* seja igual a *x*, senão `False`	(1)
`x not in s`	`False` caso um item de *s* for igual a *x*, senão `True`	(1)
`s + t`	a concatenão de *s* e *t*	(6)(7)
`s * n` ou `n * s`	equivalente a adicionar *s* a si mesmo *n* vezes	(2)(7)
`s[i]`	*i*enésimo item de *s*, origem 0	(3)
`s[i:j]`	fatia de *s* desde *i* para *j*	(3)(4)
`s[i:j:k]`	fatia de *s* desde *i* para *j* com passo *k*	(3)(5)
`len(s)`	comprimento de *s*	
`min(s)`	menor item de *s*	
`max(s)`	maior item de *s*	
`s.index(x[, i[, j]])`	índice da primeira ocorrência de *x* em *s* (no índice *i* e antes do índice *j*)	(8)
`s.count(x)`	numero total de ocorrência de *x* em *s*	

Sequências do mesmo type também suportam comparações. Em particular, tuplas e listas são comparadas lexicograficamente pela comparação de elementos correspondentes. Isso significa que para comparar igualmente, cada elemento deve comparar igual e as duas sequências devem ser do mesmo tipo e possuirem o mesmo comprimento. (Para detalhes completos veja comparisons na referência da linguagem.)

Notas:

(1) Enquanto as operações `in` e `not in` são usadas somente para ensaios de contenção simples em modo geral, algumas sequências especializadas (tais como *`str`*, *`bytes`* e *`bytearray`*) também usam eles para testing:: subsequentes.

```
>>> "gg" in "eggs"
True
```

(2) Os valores de *n* menos `0` são tratados como `0` (o que produz uma seqüência vazia do mesmo tipo que *s*). Observe que os itens na seqüência *s* não são copiados; eles são referenciados várias vezes. Isso freqüentemente assombra novos programadores Python; considere então que:

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

O que aconteceu é que `[[]] `` é uma lista de um elemento contendo uma lista vazia, então todos os três elementos de ``[[]] * 3` são referências a esta única lista vazia. Modificar qualquer um dos elementos de `lists` modifica a lista vazia. Podemos criar uma lista de listas diferentes dessa maneira:

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

Outra explicação está disponível em FAQ faq-multidimensional-list.

(3) Se *i* ou *j* forem negativo, o índice será relativo ao fim da seqüência *s*: “ len(s) + i“ ou `len(s) + j` será substituído. Mas note que `-0` ainda será `0`.

(4) A fatia *s* de *i* para *j* é definida como a seqüência de itens com índice *k* tal que `i <= k <j`. Se *i* ou *j* forem maior do que `len(s)`, use `len(s)`. Se *i* for omitido ou for igual a `None`, use `0`. Se *j* for omitido ou `None`, use `len(s)`. Se *i* for maior ou igual a *j*, a fatia está vazia.

(5) A fatia *s* de *i* para *j* com passo *k* é definida como sendo a seqüência de itens com índice `x = i + n * k` tal que `0 <= n <(j-i)/k`. Em outras palavras, os índices são `i`, `i+k`, `i+2*k`, `i+3*k` e assim por diante, parando quando *j* for atingiu (mas nunca incluindo *j*). Quando *k* for positivo, *i* e *j* serão reduzidos a `len(s)` se forem maiores. Quando *k* for negativo, *i* e *j* são reduzidos para `len(s)-1` se forem maiores. Se *i* ou *j* forem omitidos ou " None', eles se tornam valores "finais" (cujo final depende de *k*). Nota: *k* não pode ser zero. Se *k* for `None`, o mesmo será tratado como sendo igual a `1`.

(6) Concatenar sequências imutáveis sempre resulta em um novo objeto. Isso significa que a criação de uma sequência por concatenação repetida terá um custo quadrático de tempo de execução no comprimento total da seqüência. Para obter um custo de tempo de execução linear, devemos alternar para uma das alternativas abaixo:

- Se concatenar objetos *str*, podemos criar uma lista e usar *str.join()* no final ou então escrever numa instância de *io.StringIO* e recuperar o seu valor ao final
- se concatenarmos objetos *bytes*, também poderemos usar o método *bytes.join()* ou *io.BytesIO*, ou poderemos fazer concatenação in-place com um objeto *bytearray*. A classe *bytearray* são objetos mutáveis e possuem um eficiente mecanismo de overallocation
- se concatenar o objeto *tuple*, estenda a classe *list* em vez disso
- Para outros tipos, busque na documentação relevante da classe

(7) Alguns tipos de sequência (como *range*) apenas suportam sequências de itens que seguem padrões específicos e, portanto, não suportam concatenação ou repetição de sequência.

(8) `index` levanta *ValueError* quando *x* não é encontrado em *s*. Nem todas as implementações suportam a passagem dos argumentos adicionais *i* e *j*. Esses argumentos permitem a pesquisa eficiente de subseções da sequência. Passar os argumentos extras é aproximadamente equivalente a usar `s[i:j].index(x)`, apenas sem copiar nenhum dado e com o índice retornado relativo ao início da sequência e não ao início da fatia.

4.6.2 Tipos de Sequência Imutáveis

A única operação que os tipos de sequência imutáveis geralmente implementam que também não são implementada pelos tipos de sequência mutable é suporte para a função built-in *hash()*.

Esse suporte permite sequências imutáveis, tais como instâncias da classe *tuple*, para serem usadas como chaves de dicionários *dict* e armazenados em sets *set* e instâncias de *frozenset*.

A tentativa de obter um hash de uma sequência imutável que contém valores desnecessários resultará em um erro *TypeError*.

4.6.3 Tipos de Sequências Mutáveis

As operações na tabela a seguir são definidas em tipos de sequência mutáveis. A ABC *collections.abc.MutableSequence* é fornecido para tornar mais fácil a implementação correta dessas operações em tipos de sequências personalizados.

Na tabela *s* há uma instância de um tipo de sequência mutáveis, *t* é qualquer objeto iterável e *x* é um objeto arbitrário que atende a qualquer restrição de tipo e valor imposto por *s* (por exemplo *bytearray* só aceita inteiros que atendam a restrição de valor `0 <= x <= 255`).

Operação	Resultado	Notas
`s[i] = x`	item *i* de *s* é substituído por *x*	
`s[i:j] = t`	fatias de *s* de *i* para *j* são substituídas pelo conteúdo do iterável *t*	
`del s[i:j]`	o mesmo que `s[i:j] = []`	
`s[i:j:k] = t`	os elementos de `s[i:j:k]` são substituídos por aqueles de *t*	(1)
`del s[i:j:k]`	remove os elementos de `s[i:j:k]` desde a listas	
`s.append(x)`	adiciona *x* no final da sequência (igual a `s[len(s):len(s)] = [x]`)	
`s.clear()`	remove todos os itens de *s* (mesmo que `del s[:]`)	(5)
`s.copy()`	cria uma cópia rasa de *s* (mesmo que `s[:]`)	(5)
`s.extend(t)` ou `s += t`	estende *s* com o conteúdo de *t* (na maior parte do mesmo `s[len(s):len(s)] = t`)	
`s *= n`	atualiza *s* com o seu conteúdo por *n* vezes	(6)
`s.insert(i, x)`	Insere *x* dentro de *s* na posição dada por index *i* (igual a `s[i:i] = [x]`)	
`s.pop([i])`	retorna o item em *i* e também remove-o de *s*	(2)
`s.remove(x)`	remove o primeiro item de *s* sendo `s[i]` igual a *x*	(3)
`s.reverse()`	inverte os itens de *s* in-place	(4)

Notas:

(1) *t* deve ter o mesmo comprimento que a fatia que está sendo substituindo.

(2) O argumento opcional *i* é padrão para `-1`, de modo que, por padrão, o último item é removido e retornado.

(3) `remove` levanta a exceção *`ValueError`* quando *x* não for encontrado em *s*.

(4) O método `reverse()` modifica a sequência no lugar para economizar espaço ao reverter uma grande sequência. Para lembrar os usuários que isso ocorre como sendo um efeito colateral, o mesmo não retorna a sequência invertida.

(5) `clear()` e `copy()` são incluídos para consistência com as interfaces dos contêiners mutáveis que não suportam operações de fatiamento (tais como *`dict`* e *`set`*)

Novo na versão 3.3: métodos `clear()` e `copy()`.

(6) O valor *n* é um inteiro, ou um objeto implementado `__index__()`. Valores zero e negativos de *n* limparão a sequência. Os itens na sequência não são copiados; eles são referenciados várias vezes, como explicado para `s * n` sobre *Operações de Sequências Comuns*.

4.6.4 Listas

As listas são sequências mutáveis, normalmente usadas para armazenar coleções de itens homogêneos (onde o grau preciso de similaridade variará de acordo com a aplicação).

`class list`([*iterable*])

As listas podem ser construídas de várias maneiras:

- Usando um par de colchetes para denotar uma lista vazia: `[]`
- Usando suportes a colchetes, separando itens por vírgulas: `[a]`, `[a, b, c]`
- Usando uma list comprehension: `[x for x in iterable]`
- Usando o construtor de tipo: `list()` ou `list(iterable)`

O construtor produz uma lista cujos itens são iguais e na mesma ordem que os itens *iterable*'s. Os *iterable* podem ser uma sequência, um contêiner que suporte iteração ou um objeto iterador. Se *iterable* já for uma lista, uma cópia será feita e retornada, semelhante a *iterable [:] . Por exemplo, "iterable[:]*. Por exemplo, `list('abc')`

retorna `['a', 'b', 'c']` e `list( (1, 2, 3) )` retorna `[1, 2, 3]`. Se nenhum argumento for dado, o construtor criará uma nova lista vazia `[]`.

Muitas outras operações também produzem listas, incluindo a função built-in *sorted()*.

As listas implementam todo o *common* e a *mutable* operações de sequência. As listas também fornecem o seguinte método adicional:

sort (*, *key=None*, *reverse=False*)

Esse método classifica a lista in-place, usando apenas < comparações entre itens. As exceções não são suprimidas - se qualquer operação de comparação falhar, toda a operação de ordenação falhará (e a lista provavelmente será deixada num estado parcialmente modificado).

sort() aceita 2 argumetnos que só podem ser passados como keywords (*keyword-only arguments*):

A *key* especifica uma função de um argumento que é usado para extrair uma chave de comparação de cada elemento de lista (por exemplo, `key=str.lower`). A chave correspondente a cada item na lista é calculada uma vez e depois usada para todo o processo de classificação. O valor padrão `None` significa que os itens da lista são classificados diretamente sem calcular um valor de chave separado.

A função utilitária *functools.cmp_to_key()* está disponível para converter a função *cmp* no estilo 2.x para uma função *key*.

reverse é um valor booleano. Se definido igual a `True`, então os elementos da lista são classificados como se cada comparação fosse reversa.

Este método modifica a seqüência in-place para a economizar espaço ao classificar uma grande sequência. Para lembrar aos usuários que os mesmos operam por efeito colateral, ele não retorna a seqüência ordenada (utilize a função *sorted()* para solicitar explicitamente uma nova instância da lista ordenada).

O método *sort()* é garantido para ser estável. Um tipo é estável se garante que não alterar a ordem relativa de elementos que comparam igual - isso é útil para classificar em várias passagens (por exemplo, classificar por departamento, depois por nota salarial).

CPython implementation detail: No momento em que uma lista está sendo ordenada, o efeito de tentar alterar, ou mesmo inspecionar, a lista é indefinida. A implementação C do Python faz com que a lista apareça vazia durante o tempo de processamento, e levanta a exceção *ValueError* se puder detectar que a lista foi alterada durante uma ordenação.

4.6.5 Tuplas

As Tuples são sequências imutáveis, tipicamente usadas para armazenar coleções de dados heterogêneos (como as 2-tuplas produzidas pelo função built-in *enumerate()*). As Tuples também são usadas para casos em que seja necessária uma seqüência imutável de dados homogêneos (como permitir o armazenamento em uma instância *set* ou *dict*).

class tuple ([*iterable*])

As Tuples podem ser construídos de várias maneiras:

- Usando um par de parênteses para denotar que a tupla está vazia: `()`
- Usando uma vírgula à direita para uma tupla contendo um único elemento: `a,` ou `(a,)`
- Separando os itens por vírgula: `a, b, c` ou `(a, b, c)`
- Usando a função built-in *tuple()*: `tuple()` ou `tuple(iterable)`

O construtor produz uma tupla cujos itens são iguais e na mesma ordem que os elementos *iteráveis*. Os *iterable* poderão ser uma sequência, um contêiner que suporta iteração ou um objeto iterable. Se *iterable* já for uma tupla, o mesmo será retornado inalterado. Por exemplo, `tupla('abc')` retorna `('a', 'b', 'c')` e `tupla([1, 2, 3])` retorna `(1, 2, 3)`. Se nenhum argumento for dado, o construtor criará uma tupla vazia, `()`.

Observe que na verdade é a vírgula que faz uma tupla, e não os parênteses. Os parênteses são opcionais, exceto no caso de tupla vazia, ou quando são necessários para evitar ambiguidades sintáticas. Por exemplo, `f(a, b, c)` é uma chamada da função com três argumentos, enquanto que `f((a, b, c))` é uma chamada de função com uma 3-tupla com um única argumento.

As tuplas implementam todas as operações de sequência *common*.

Para coleções heterogêneas de dados onde o acesso pelo nome é mais claro do que o acesso pelo índice, *collections.namedtuple()* pode ser uma escolha mais apropriada do que um simples objeto tupla.

4.6.6 Ranges

O tipo da classe *range* representa uma sequência imutável de números e é comumente usada para fazer um looping um número determinado de vezes por `for`.

class range(*stop*)
class range(*start*, *stop*[, *step*])

Os argumentos para o construtor de intervalo devem ser inteiros (built-ins *int* ou qualquer objeto que implemente o método especial `__index__`). Se o argumento *step* for omitido, será usado o padrão `1`. Se o argumento *start* for omitido, será usado o padrão `0`. Se *step* for zero, uma exceção :exc: *ValueError* será levantada.

Para um *step* positivo, o conteúdo de um intervalo `r` será determinado pela fórmula `r[i] = start + step*i` onde `i >= 0` and `r[i] < stop`.

Para um *passo*, o conteúdo do intervalo ainda será determinado pela fórmula `r[i] = start + step*i`, mas as restrições serão `i >= 0` and `r[i] > stop`.

Um objeto range estará vazio se `r[0]` não atender a restrição de valor. Os Ranges suportam índices negativos, mas estes são interpretados como indexadores partindo do final da seqüência determinada pelos índices positivos.

Ranges contendo valores absolutos maiores do que *sys.maxsize* são permitidos, mas alguns recursos (como *len()*) podem levantar uma exceção *OverflowError*.

Range de exemplos:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

Os Ranges todas as operações de sequência *common* exceto a concatenção e a repetição (devido ao fato de que os objetos Range só podem representar sequências que seguem um padrão rígido e a repetição e a concatenação geralmente violam esse padrão).

start
O valor do parâmetro *start* (ou `0` se o parâmetro não for fornecido)

stop
O valor do parâmetro *stop*

step
O valor do parâmetro *step* (ou `1` se o parâmetro não for fornecido)

A vantagem da classe *range* sobre o tipo regular *list* ou o *tuple* é que um objeto *range* sempre terá a mesma quantidade (pequena) de memória, não importa o tamanho do intervalo o mesmo esteja representando (como ele apenas armazena os valores `start`, `stop` e `step`, calculando todos os demais itens individualmente e gerando os subranges conforme necessário).

Objetos Range implementam a classe *collections.abc.Sequence* ABC, e fornecem recursos como testes de contenção, pesquisa de índice de elemento, fatiamento e suporte a índices negativos (veja *Tipos de Sequências — list, tuple, range*):

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

Testar objetos Range com o sinal de igualdade == e `!=` os compara como sequências. Ou seja, dois objetos de intervalo são considerados iguais se representarem a mesma sequência de valores. (observe que dois objetos Range que comparam igual podem ter diferentes atributos *start*, *stop* and *step*, por exemplo `range(0) == range(2, 1, 3)` ou `range(0, 3, 2) == range(0, 4, 2)`.)

Alterado na versão 3.2: Implementar a Sequencia ABC. Suporte a fatiamento e a índices negativos. Testa objetos *int* para associação em tempo constante em vez de iterar através de todos os itens.

Alterado na versão 3.3: Determina '==' e '!=' para comparar objetos Range com base na sequência de valores que eles definem (em vez de comparar com base na identidade do objeto).

Novo na versão 3.3: Os atributos *start*, *stop* e *step*.

Ver também:

- A receita de linspace mostra como implementar uma versão preguiçosa de um intervalo adequado para aplicações de ponto flutuante.

4.7 Tipo de Sequência de Texto — `str`

Os dados textuais em Python são tratados com objetos *str*, ou *strings*. Strings são imutáveis *sequences* de códigos Unicode. As Strings literais são escritas de diversas maneiras:

- Aspas Simples: `'allows embedded "double" quotes'`
- Aspas Duplas: `"allows embedded 'single' quotes"`.
- Aspas Triplas: `'''Three single quotes'''`, `"""Three double quotes"""`

Aspas triplas são Strings de várias linhas - todos os espaços em branco associados serão incluídos na String literal.

Os literais Strings que fazem parte de uma única expressão e têm apenas espaços em branco entre eles serão implicitamente convertidos em um único literal String. Isso é, `("spam " "eggs") == "spam eggs"`.

Veja strings para mais informações sobre as várias formas de cadeia de literais, incluindo o suporte a Strings escape, e o prefixo `r` ("raw") que desabilita a maioria dos processos de escape.

As Strings também podem ser criadas a partir de outros objetos usando o construtor *str*.

Uma vez que não há nenhum tipo de "caractere" separador, indexar uma String produz Strings de comprimento 1. Ou seja, para uma Strings não vazia *s*, `s[0] == s[0:1]`.

Também não existe um tipo de String mutável, mas o método *str.join()* ou a classe :class:' io.StringIO' podem ser usado para construir Strings de forma eficiente a partir de vários partes distintas.

Alterado na versão 3.3: Para a compatibilidade com versões anteriores do Python 2, o prefixo `u` for mais uma vez permitido em literais Strings. Não possui quaisquer efeito sobre o significado de literais Strings e não pode ser combinado com o prefixo `r`.

class str(*object=''*)
class str(*object=b''*, *encoding='utf-8'*, *errors='strict'*)

Retorna uma *string* versão do *object*. Se o *object* não é fornecido, retorna uma String vazia. Caso contrário, o comportamento de `str()` dependerá se o *encoding* ou *erros* são fornecidos, da seguinte forma.

Se nem o *encoding* nem os *errors* forem dados, a `str(object)` retorna o método `object.__str__()`, que é a representação de seqüência "informal" ou que pode ser facilmente imprimível de *objeto*. Para objetos String, esta é a própria String. Se o *objeto* não tiver um método `__str__()`, então a função *str()* retornará `repr(object)()`.

Se pelo menos um de *encoding* ou *errors* for fornecido, *object* deve ser um *bytes-like object* (por exemplo. *bytes* ou *bytearray*). Nesse caso, se *object* for um objeto *bytes* (ou *bytearray*), então `str(bytes, encoding, errors)` será equivalente a *bytes.decode(encoding, errors)*. Caso contrário, o objeto bytes subjacente ao objeto buffer é obtido antes de chamar *bytes.decode()*. Veja *Tipos de Sequência Binária — bytes, bytearray, memoryview* e bufferobjects para obter informações sobre objetos buffer.

Passa um objeto *bytes* para *str()* sem os argumentos *encoding* ou *errors* se enquadra no primeiro caso de retornar a representação informal de strings (consulte também a opção de linha **:option:command:'-b'** para Python). Por exemplo:

```
>>> str(b'Zoot!')
"b'Zoot!'"
```

Para mais informações sobre a classe `str` e seus métodos, veja *Tipo de Sequência de Texto — str* e a seção *Métodos de String* abaixo. Para gerar strings formatadas, veja as seções f-strings e *Sintaxe das strings de formato*. Além disso, veja a seção *Serviços de Processamento de Texto*.

4.7.1 Métodos de String

Strings implementam todas as operações de sequência *common*, juntamente com os métodos adicionais descritos abaixo.

Strings também suportam duas formas de formatação de string, uma fornecendo uma ampla gama de flexibilidade e customização (veja *str.format()*, *Sintaxe das strings de formato* e *Formatação personalizada de strings*) e a outra baseada no estilo de formatação `printf` da linguagem C, que lida com uma possibilidade menor de tipos e é levemente mais difícil de utilizar corretamente, mas é frequentemente mais rápida para os casos na qual ela consegue lidar (*Formatação de String no Formato printf-style*).

A seção *Serviços de Processamento de Texto* da biblioteca padrão cobre um número de diversos outros módulos que fornecem vários utilitários relacionados a texto (incluindo suporte a expressões regulares no módulo *re*).

str.**capitalize**()
Retorna uma cópia da String com o seu primeiro caractere em maiúsculo e o restantes em minúsculo.

str.**casefold**()
Retorna uma cópia da string em casefolded. Strings em casefold podem ser usadas para corresponder letras sem importar se são minúsculas/maiúsculas.

Casefolding é similar a mudar para letras minúsculas, mas mais agressivo porque ele é pretendido para remover todas as diferenças maiúsculas/minúsculas em uma string. Por exemplo, a letra minúscula alemã `'ß'` é equivalente a `"ss"`. Como ela já é uma minúscula, o método *lower()* não irá fazer nada para `'ß'`; já o método *casefold()* converte a letra para `"ss"`.

O algoritmo casefolding é descrito na seção 3.13 do Padrão Unicode.

Novo na versão 3.3.

str.**center**(*width*[, *fillchar*])
Retorna um texto centralizado em uma string de comprimento *width*. Preenchimento é feito usando o parâmetro *fillchar* especificado (padrão é o caractere de espaço ASCII). A string original é retornada se *width* é menor ou igual que `len(s)`.

str.**count**(*sub*[, *start*[, *end*]])
Retorna o número de ocorrências da sub-string *sub* que não se sobrepõem no intervalo [*start*, *end*]. Argumentos opcionais *start* e *end* são interpretados como na notação de fatias.

str.**encode**(*encoding="utf-8"*, *errors="strict"*)
Retornar uma versão codificada da String como um objeto bytes. A codificação padrão é `'utf-8'`. *erros* podem ser levantados para definir um esquema de tratamento de erros diferente. O padrão para *erros* é `'strict'`, o que significa que os erros de codificação levantam uma exceção *UnicodeError*. Outros valores possíveis são `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` e qualquer outro nome registrado via *codecs.register_error()*, veja a seção *Error Handlers*. Para obter uma lista das possíveis codificações, consulte a seção *Standard Encodings*.

Alterado na versão 3.1: Suporte para argumentos que possuem keyword adicionados.

str.**endswith**(*suffix*[, *start*[, *end*]])
Retorna `True` se a string terminar com o *suffix* especificado, caso contrário retorna `False`. *suffix* também pode ser uma tupla de sufixos para procurar. Com o parâmetro opcional *start*, começamos a testart a partir daquela posição. Com o parâmetro opcional *end*, devemos parar de comparar na posição especificada.

str.**expandtabs**(*tabsize=8*)
Devolve uma cópia da string onde todos os caracteres de tabulação são substituídos por um ou mais espaços, dependendo da coluna atual e do tamanho fornecido para a tabulação. Posições de tabulação ocorrem a cada *tabsize* caracteres (o padrão é 8, dada as posições de tabulação nas colunas 0, 8, 16 e assim por diante). Para expandir a string, a coluna atual é definida como zero e a string é examinada caracter por caracter. Se o caracter é uma tabulação (`\t`), um ou mais caracteres de espaço são inseridos no resultado até que a coluna atual seja igual a próxima posição de tabulação. (O caracter de tabulação em si não é copiado.) Se o caracter é um caracter de nova linha (`\n`) ou de retorno (`\r`), ele é copiado e a coluna atual é resetada para zero. Qualquer outro caracter é copiado sem ser modificado e a coluna atual é incrementada em uma unidate independentemente de como o caracter é representado quanto é impresso.

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012     0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
```

str.**find**(*sub*[, *start*[, *end*]])
Retorna o índice mais baixo na String onde a substring *sub* é encontrado dentro da fatia `s[start:end]`. Argumntos opcionais como *start* e *end* são interpretados como umanotação de fatiamento. Retorna `-1` se *sub* não for

localizado.

Nota: O método *find()* deve ser usado apenas se precisarmos conhecer a posição de *sub*. Para verificar se *sub* é ou não uma substring, use o operador `in`:

```
>>> 'Py' in 'Python'
True
```

---

str.**format**(**args*, ***kwargs*)

Executa uma operação de formatação de string. A string na qual este método é chamado pode conter texto literal ou campos para substituição delimitados por chaves `{}`. Cada campo de substituição contém ou um índice numérico de um argumento posicional, ou o nome de um argumento palavra-chave. Retorna a cópia da string onde cada campo para substituição é substituído com o valor da string do argumento correspondente.

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

Veja *Sintaxe das strings de formato* para uma descrição das várias opções de formatação que podem ser especificadas em um formato Strings.

Nota: Ao formatar um número (*int*, *float*, *complex*, *decimal.Decimal* e sub-classes) com o tipo `n` (ex: `'{:n}'.format(1234)`), a função define temporariamente a localidade `LC_CTYPE` para a localidade `LC_NUMERIC` a fim de decodificar campos `decimal_point` e `thousands_sep` de `localeconv()` se eles são caracteres não-ASCII ou maiores que 1 byte, e a localidade `LC_NUMERIC` é diferente da localidade `LC_CTYPE`. Esta mudança temporária afeta outras threads.

Alterado na versão 3.7: Ao formatar um número com o tipo `n`, a função define temporatiamente a localidade `LC_CTYPE` para `LC_NUMERIC` em alguns casos.

str.**format_map**(*mapping*)

Semelhante a `str.format(**mapping)`, exceto pelo fato `mapping` de que é usado indiretamente e não copiado para uma classe *dict*. Isso é útil se, por exemplo, "mapping" é uma subclasse de dict:

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

Novo na versão 3.2.

str.**index**(*sub*[, *start*[, *end*]])

Similar a *find()*, mas levanta *ValueError* quando a sub-string não é encontrada.

str.**isalnum**()

Retorna `True` se todos os caracteres na string são alfanuméricos e existe pelo menos um caractere, ou `False` caso contrário. Um caractere `c` é alfanumérico se um dos seguintes retorna `True`: `c.isalpha()`, `c.isdecimal()`, `c.isdigit()`, ou `c.isnumeric()`.

str.**isalpha**()

Retorna `True` se todos os caracteres na string são alfabéticos e existe pelo menos um caractere, `False` caso contrário. Caracteres alfabéticos são aqueles caracteres definidos na base de dados de caracteres Unicode como "Letra", isto é, aqueles cuja propriedade na categoria geral é um destes: "Lm", "Lt", "Lu", "Ll", ou "Lo". Perceba que isso é diferente da propriedade "Alfabética" definida no Unicode padrão.

str.**isascii**()
Retorna `True` se a string é vazia ou se todos os caracteres na string são ASCII, `False` caso contrário. Caracteres ASCII tem códigos inteiros no intervalo U+0000-U+007F.

Novo na versão 3.7.

str.**isdecimal**()
Retorna `True` se todos os caracteres na string são caracteres decimais e existe pelo menos um caractere, `False` caso contrário. Caracteres decimais são aqueles que podem ser usados para formar números na base 10, exemplo U+0660, ou dígito zero para arábico-índico. Formalmente, um caractere decimal é um caractere em Unicode cuja categoria geral é "Nd".

str.**isdigit**()
Retorna `True` se todos os caracteres na string são dígitos e existe pelo menos um caractere, `False` caso contrário. Dígitos incluem caracteres decimais e dígitos que precisam de tratamento especial, tal como a compatibilidade com dígitos sobre-escritos. Isso inclui dígitos que não podem ser usados para formar números na base 10, como por exemplo os números de Kharosthi. Formalmente, um dígito é um caractere que tem a propriedade com valor Numeric_Type=Digit ou Numeric_Type=Decimal.

str.**isidentifier**()
Retorna `True` se a string é um identificador válido conforme a definição da linguagem, seção identifiers.

Use *keyword.iskeyword()* to test for reserved identifiers such as `def` and `class`.

str.**islower**()
Retorna `True` se todos os caracteres em caixa[4] na string são minúsculos e existe pelo menos um caracter em caixa, `False` caso contrário.

str.**isnumeric**()
Retorna `True` se todos os caracteres na string são caracteres numéricos, e existe pelo menos um caractere, `False` caso contrário. Caracteres numéricos incluem dígitos, e todos os caracteres que tem a propriedade/valor numérica Unicode, isto é: U+2155, um quinto de fração vulgar. Formalmente, caracteres numéricos são aqueles que possuem propriedades com valor Numeric_Type=Digit, Numeric_Type=Decimal ou Numeric_Type=Numeric.

str.**isprintable**()
Retorna `True` se todos os caracteres na string podem ser impressos ou se a string é vazia, `False` caso contrário. Caracteres que não podem ser impressos são aqueles que estão definidos no banco de dados Unicode como "Outros" ou "Separadores", exceto o caractere ASCII que representa o espaço (0x20), o qual é impresso. (Perceba que caracteres que podem ser impressos, neste contexto, são aqueles que não devem ser escapados quando *repr()* é invocada sobre uma string. Ela não tem sentido no tratamento de strings escritas usando *sys.stdout* ou *sys.stderr*.)

str.**isspace**()
Retorna `True` se existem apenas caracteres em branco na string e existe pelo menos caractere, `False` caso contrário.

Um caracter é *espaço em branco* se no banco de dados de caracteres Unicode (veja *unicodedata*), ou pertence a categoria geral `Zs` ("Separador, espaço"), ou dua classe bidirecional é `WS`, `B`, ou `S`.

str.**istitle**()
Retorna `True` se a string é titlecased e existe pelo menos um caractere, por exemplo caracteres maiúsculos somente podem proceder caracteres que não diferenciam maiúsculas/minúsculas, e caracteres minúsculos somente podem proceder caracteres que permitem ambos. Retorna `False` caso contrário.

str.**isupper**()
Retorna `True` se todos os caracteres que permitem maiúsculas ou minúsculas[4] na string estão com letras maiúsculas, e existe pelo menos um caractere maiúsculo, `False` caso contrário.

[4] Caracteres que possuem maiúsculo e minúsculo são aqueles com a propriedade de categoria geral igual a "Lu" (Letra, maiúscula), "Ll" (Letra, minúscula), ou "Lt" (Letra, em formato de título).

str.**join**(*iterable*)
Retorna a string que é a concatenação das strings no *iterável*. Um *TypeError* será levantado se existirem quaisquer valores que não sejam strings no *iterável*, incluindo objetos *bytes*. O separador entre elementos na é a string que está fornecendo este método.

str.**ljust**(*width*[, *fillchar*])
Retorna a string alinhada a esquerda em uma string de comprimento *width*. Preenchimento é feito usando *fillchar* que for especificado (o padrão é o caractere ASCII de espaço). A string original é retornada se *width* é menor ou igual que `len(s)`.

str.**lower**()
Retorna uma cópia da string com todos os caracteres que permitem maiúsculo E minúsculo[4] convertidos para letras minúsculas.

O algoritmo usado para letras minúsculas é descrito na seção 3.13 do Padrão Unicode.

str.**lstrip**([*chars*])
Retorna uma cópia da string com caracteres iniciais removidos. O argumento *chars* é uma string que especifica o conjunto de caracteres a serem removidos. Se for omitido ou se for `None`, o argumento *chars* será considerado como espaço em branco por padrão para a remoção. O argumento *chars* não é um prefixo; ao invés disso, todas as combinações dos seus valores são retirados:

```
>>> '   spacious   '.lstrip()
'spacious   '
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

static str.**maketrans**(*x*[, *y*[, *z*]])
Este método estático retorna uma tabela de tradução usável para *str.translate()*.

Se existe apenas um argumento, ele deve ser um dicionário mapeando números Unicode (inteiros) ou caracteres (strings de comprimento 1) para números Unicode, strings (de comprimento arbitrário) ou `None`. Caracteres chave serão então convertidos para números ordinais.

Se existirem dois argumentos, eles devem ser strings de igual tamanho, e no dicionário resultante, cada caractere em x será mapeado para o caractere na mesma posição em y. Se existir um terceiro argumento, ele deve ser uma string, cujos caracteres serão mapeados para `None` no resultado.

str.**partition**(*sep*)
Quebra a string na primeira ocorrência de *sep*, e retorna uma tupla com 3 elementos contendo a parte antes do separador, o próprio separator, e a parte após o separador. Se o separador não for encontrado, retorna uma tupla com 3 elementos contendo a string, seguido de duas strings vazias.

str.**replace**(*old*, *new*[, *count*])
Retorna uma cópia da string com todas as ocorrências da substring *old* substituídas por *new*. Se o argumento opcional *count* é fornecido, apenas as primeiras *count* ocorrências são substituídas.

str.**rfind**(*sub*[, *start*[, *end*]])
Retorna o maior índice onde a substring *sub* foi encontrada dentro da string, onde *sub* está contida dentro do intervalo `s[start:end]`. Argumentos opcionais *start* e *end* são interpretados usando a notação slice. Retorna `-1` em caso de falha.

str.**rindex**(*sub*[, *start*[, *end*]])
Similar a *rfind()* mas levanta um *ValueError* quando a substring *sub* não é encontrada.

str.**rjust**(*width*[, *fillchar*])
Retorna a string alinhada à direita em uma string de comprimento *width*. Preenchimento é feito usando o caractere *fillchar* especificado (o padrão é um caractere de espaço ASCII). A string original é retornada se *width* é menor que ou igual a `len(s)`.

str.**rpartition**(*sep*)
Quebra a string na última ocorrência de *sep*, e retorna uma tupla com 3 elementos contendo a parte antes do separador, o próprio separator, e a parte após o separador. Se o separador não for encontrado, retorna uma tupla com 3 elementos contendo duas strings vazias, seguido da própria string original.

str.**rsplit**(*sep=None*, *maxsplit=-1*)
Retorna uma lista de palavras na string, usando *sep* como a string delimitadora. Se *maxsplit* é fornecido, no máximo *maxsplit* cortes são feitos, sendo estes mais *à direita*. Se *sep* não foi especificado ou `None` foi informado, qualquer string de espaço em branco é um separador. Exceto pelo fato de separar pela direita, *`rsplit()`* se comporta como *`split()`*, o qual é descrito em detalhes abaixo.

str.**rstrip**([*chars*])
Retorna uma cópia da string com caracteres no final removidos. O argumento *chars* é uma string que especifica o conjunto de caracteres para serem removidos. Se omitidos ou tiver o valor `None`, o argumento *chars* considera como padrão a remoção dos espaços em branco. O argumento *chars* não é um sufixo; ao invés disso, todas as combinações dos seus valores são removidos:

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

str.**split**(*sep=None*, *maxsplit=-1*)
Retorna uma lista de palavras na string, usando *sep* como a string delimitadora. Se *maxsplit* é fornecido, no máximo *maxsplit* cortes são feitos (portando, a lista terá no máximo `maxsplit+1` elementos). Se *maxsplit* não foi especificado ou `-1` foi informado, então não existe limite no número de cortes (todos os cortes possíveis são realizados).

Se *sep* é fornecido, delimitadores consecutivos não são agrupados juntos e eles são destinados para delimitar strings vazias (por exemplo `'1,,2'.split(',')` retorna `['1', '', '2']`). O argumento *sep* pode consistir de múltiplos caracteres (por exemplo, `'1<>2<>3'.split('<>')` retorna `['1', '2', '3']`). Separar uma string vazia com um separador especificado retorna `['']`.

Por exemplo:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
```

Se *sep* não for especificado ou for `None`, um algoritmo diferente de separação é aplicado: ocorrências consecutivas de espaços em branco são consideradas como um separador único, e o resultado não conterá strings vazias no início ou no final, se a string tiver espaços em branco no início ou no final. Consequentemente, separar uma string vazia ou uma string que consiste apenas de espaços em branco com o separador `None`, retorna `[]`.

Por exemplo:

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> '   1   2   3   '.split()
['1', '2', '3']
```

str.**splitlines**([*keepends*])
Retorna uma lista das linhas na string, quebrando a mesma nas fronteiras de cada linha. Quebras de linhas não são incluídas na lista resultante a não ser que *keepends* seja fornecido e seja verdadeiro.

Este método divide nos seguintes limites das linhas. Em particular, os limites são um superconjunto de *universal newlines*.

Representação	Description (descrição)
`\n`	Feed de linha
`\r`	Retorno de Carro
`\r\n`	Retorno do Carro + Feed da Linha
`\v` ou `\x0b`	Tabulação de Linha
`\f` ou `\x0c`	Formulário de Feed
`\x1c`	Separador de Arquivos
`\x1d`	Separador de Grupo
`\x1e`	Separador de Registro
`\x85`	Próxima Linha (C1 Control Code)
`\u2028`	Separador de Linha
`\u2029`	Parágrafo Separador

Alterado na versão 3.2: `\v` e `\f` adicionado à lista de limites de linha.

Por exemplo:

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

Ao contrário do método *`split()`* quando um delimitador de String *sep* é fornecido, este método retorna uma lista vazia para a uma String vazia e uma quebra de linha de terminal não resulta numa linha extra:

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

Para comparação, temos `split('\n')`:

```
>>> ''.split('\n')
['']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

`str.`**`startswith`**(*prefix*[, *start*[, *end*]])

Retorne `True` se a String começar com o *prefixo*, caso contrário, retorna `False`. *prefixo* também pode ser uma tupla de prefixos a serem procurados. Com *start* opcional, a String de teste começa nessa posição. Com *fim* opcional, interrompe a comparação de String nessa posição.

`str.`**`strip`**([*chars*])

Retorna uma cópia da string com caracteres no início e no final removidos. O argumento *chars* é uma string que especifica o conjunto de caracteres a serem removidos. Se for omitido ou for `None`, o argumento *chars* irá remover por padrão os caracteres em branco. O argumento *chars* não é um prefixo, nem um sufixo; ao contrário disso, todas as combinações dos seus seus valores são removidas:

```
>>> '   spacious   '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

Os valores do argumento *chars* são removidos dos extremos inicial e final da string. Caracteres são removidos do extremo inicial até atingir um caractere da string que não está contido no conjunto de caracteres em *chars*. Uma ação similar acontece no extremo final da string. Por exemplo:

```
>>> comment_string = '#....... Section 3.2.1 Issue #32 .......'
>>> comment_string.strip('.#! ')
'Section 3.2.1 Issue #32'
```

`str.`**`swapcase`**`()`
Retorna uma cópia da string com caracteres maiúsculos convertidos para minúsculos e vice-versa. Perceba que não é necessariamente verdade que `s.swapcase().swapcase() == s`.

`str.`**`title`**`()`
Retorna uma versão titlecased da string, onde palavras iniciam com um caractere com letra maiúscula e os caracteres restantes são em letras minúsculas.

Por exemplo:

```
>>> 'Hello world'.title()
'Hello World'
```

O algoritmo usa uma definição simples independente de idioma para uma palavra, como grupos de letras consecutivas. A definição funciona em muitos contextos, mas isso significa que apóstrofes em contradições e possessivos formam limites de palavras, os quais podem não ser o resultado desejado:

```
>>> "they're bill's friends from the UK".title()
"They're Bill's Friends From The Uk"
```

Uma solução alternativa para os apóstrofes pode ser construída usando expressões regulares:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

`str.`**`translate`**`(`*table*`)`
Retorna uma cópia da string na qual cada caractere foi mapeado através da tabela de tradução fornecida. A tabela deve ser um objeto que implementa indexação através de `__getitem__()`, tipicamente um *mapeamento* ou uma *sequência*. Quando indexada por um ordinal unicode (um inteiro), o objeto tabela pode fazer qualquer uma das seguintes ações: retornar um ordinal unicode ou uma string, para mapear o caractere para um ou mais caracteres; retornar `None`, para deletar o caractere da string de retorno; ou levantar uma exceção *`LookupError`*, para mapear o caractere para si mesmo.

Você pode usar *`str.maketrans()`* para criar um mapa de tradução com mapeamentos caractere para caractere em diferentes formatos.

Veja também o módulo *`codecs`* para uma abordagem mais flexível para mapeamento de caracter customizado.

`str.`**`upper`**`()`
Retorna uma cópia da string com todos os caracteres que permitem maiúsculo e minúsculo[4] convertidos para letras maiúsculas. Perceba que `s.upper().isupper()` pode ser `False` se `s` contiver caracteres que não possuem maiúsculas e minúsculas, ou se a categoria Unicode do(s) caractere(s) resultante(s) não for “Lu” (Letra maiúscula), mas por ex “Lt” (Letra em titlecase).

O algoritmo de maiúsculas utilizado é descrito na seção 3.13 do Padrão Unicode.

`str.`**`zfill`**`(`*width*`)`
Retorna uma cópia da String deixada preenchida com dígitos ASCII `'0'` para fazer uma string de comprimento *width*. Um prefixo sinalizador principal (`'+'`/`'-'`) será tratado inserindo o preenchimento *após* o caractere de sinal em vez de antes. A String original será retornada se o *width* for menor ou igual a `len(s)`.

Por exemplo:

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

4.7.2 Formatação de String no Formato `printf`-style

Nota: As operações de formatação descritas aqui exibem uma variedade de peculiaridades que levam a diversos erros comuns (tais como não conseguir exibir tuplas e dicionários corretamente). Usar o novo formatador de strings literais (f-strings), a interface *`str.format()`*, ou *templates de strings* pode ajudar a evitar esses erros. Cada uma dessas alternativas fornece seus próprios custos e benefícios de simplicidade, flexibilidade, e/ou extensibilidade.

Os objetos String possuem um única operador built-in: o operador `%` ``(modulo)``. O mesmo também é conhecido como o *formatador* de String ou como operador *interpolador*. Dado ``format % values`` (onde *format* é uma String), as especificações de conversão `%` em *format* são substituídas por zero ou mais elementos de *valores*. O efeito é semelhante ao uso da função `sprintf()` na linguagem C.

Se *format* precisar de um único operador, *valores* podem ser objetos simples ou que não sejam uma tupla.[5] Caso contrário, *valores* precisarão ser uma tupla com exatamente o número de itens especificados pela string de formatação, ou um único mapa de objetos (por exemplo, um dicionário).

Um especificador de conversão contém dois ou mais caracteres e tem os seguintes componentes, que devem aparecer nesta ordem:

1. O caractere `'%'`, que determina o início do especificador.
2. Mapeamento de Chaves (opcional), consistindo de uma sequência entre parênteses de caracteres (por exemplo, `(algumnome)`).
3. Flags de conversão (opcional), que afetam o resultado de alguns tipos de conversão.
4. Largura mínima do Campo(opcional). Se for especificado por `'*'` (asterisco), a largura real será lida a partir do próximo elemento da tupla em *values* e o objeto a converter virá após a largura mínima do campo e a precisão que é opcional.
5. Precisão (opcional), fornecido como uma `'.'` (ponto) seguido pela precisação. Se determinado como um `'*'` (um asterisco), a precisão real será lida a partir do próximo elemento da tupla em *values*, e o valor a converter virá após a precisão.
6. Modificador de Comprimento(opcional).
7. Tipos de Conversão

Quando o argumento certo for um dicionário (ou outro tipo de mapeamento), então os formatos na string *deverão* incluir uma chave de mapeamento entre parênteses nesse dicionário inserido imediatamente após o caractere `'%'`. A key de mapeamento seleciona o valor a ser formatado a partir do mapeamento. Por exemplo:

[5] Para formatar apenas uma tupla, você deve portanto fornecer uma tupla singleton, cujo único elemento é a tupla a ser formatada.

```
>>> print('%(language)s has %(number)03d quote types.' %
...       {'language': "Python", "number": 2})
Python has 002 quote types.
```

Nesse caso, nenhum especificador * poderá ocorrer num formato (uma vez que eles exigem uma lista de parâmetros sequenciais).

Os caracteres flags de conversão são:

Flag	Significado
'#'	A conversão de valor usará o "formulário alternativo" (em que definimos abaixo).
'0'	A conversão será preenchida por zeros para valores numéricos.
'-'	O valor convertido será ajustado à esquerda (substitui a conversão '0' se ambos forem fornecidos).
' '	(um espaço) Um espaço em branco deverá ser deixado antes de um número positivo (ou uma String vazia) produzido por uma conversão assinada.
'+'	Um sinal de caractere ('+' ou '-') precedera a conversão (substituindo o sinalizador "space").

Um modificador de comprimento (h, l, ou L) pode estar presente, mas será ignorado, pois o mesmo não é necessário para o Python – então por exemplo %ld é idêntico a %d.

Os tipos de conversão são:

Conversão	Significado	Notas
'd'	Número decimal inteiro sinalizador.	
'i'	Número decimal inteiro sinalizador.	
'o'	Valor octal sinalizador.	(1)
'u'	Tipo obsoleto - é idêntico a 'd'.	(6)
'x'	Sinalizador hexadecimal (minúsculas).	(2)
'X'	Sinalizador hexadecimal (maiúscula).	(2)
'e'	Formato exponencial de ponto flutuante (minúsculas).	(3)
'E'	Formato exponencial de ponto flutuante (maiúscula).	(3)
'f'	Formato decimal de ponto flutuante.	(3)
'F'	Formato decimal de ponto flutuante.	(3)
'g'	O formato de ponto flutuante. Usa o formato exponencial em minúsculas se o expoente for inferior a -4 ou não inferior a precisão, formato decimal, caso contrário.	(4)
'G'	Formato de ponto flutuante. Usa o formato exponencial em maiúsculas se o expoente for inferior a -4 ou não inferior que a precisão, formato decimal, caso contrário.	(4)
'c'	Caráter único (aceita inteiro ou um único caractere String).	
'r'	String (converte qualquer objeto Python usando a função *repr()*).	(5)
's'	String (converte qualquer objeto Python usando a função *str()*).	(5)
'a'	String (converte qualquer objeto Python usando a função *ascii()*).	(5)
'%'	Nenhum argumento é convertido, resultando um caractere '%' no resultado.	

Notas:

(1) A forma alternativa faz com que um especificador octal principal ('0o') seja inserido antes do primeiro dígito.

(2) O formato alternativo produz um '0x' ou '0X' (dependendo se o formato 'x' or 'X' foi usado) para ser inserido antes do primeiro dígito.

(3) A forma alternativa faz com que o resultado sempre contenha um ponto decimal, mesmo que nenhum dígito o siga.

A precisão determina o número de dígitos após o ponto decimal e o padrão é 6.

(4) A forma alternativa faz com que o resultado sempre contenha um ponto decimal e os zeros à direita não sejam removidos, como de outra forma seriam.

A precisão determina o número de dígitos significativos antes e depois do ponto decimal e o padrão é 6.

(5) Se a precisão for `N`, a saída será truncada em caracteres `N`.

(6) Veja **PEP 237**.

Como as Strings do Python possuem comprimento explícito, `%s` as conversões não assumem que `'\0'` é o fim da string.

Alterado na versão 3.1: `%f` as conversões para números cujo valor absoluto é superior a 1e50 não são mais substituídas pela conversão `%g`.

4.8 Tipos de Sequência Binária — `bytes`, `bytearray`, `memoryview`

Os principais tipos embutidos para manipular dados binários são *bytes* e *bytearray*. Eles são suportados por *memoryview* a qual usa o buffer protocol para acessar a memória de outros objetos binários sem precisar fazer uma cópia.

O módulo *array* suporta armazenamento eficiente de tipos de dados básicos como inteiros de 32 bits e valores de ponto-flutuante com precisão dupla IEEE754.

4.8.1 Objetos Bytes

Objetos bytes são sequências imutáveis de bytes simples. Como muitos protocolos binários importantes são baseados em codificação ASCII de texto, objetos bytes oferecem diversos métodos que são válidos apenas quando trabalhamos com dados compatíveis com ASCII, e são proximamente relacionados com objetos string em uma variedade de outros sentidos.

class bytes([*source*[, *encoding*[, *errors*]]])

Em primeiro lugar, a sintaxe para literais de bytes é em grande parte a mesma para literais de String, exceto que um prefixo `b` é adicionado:

- Aspas simples: `b'still allows embedded "double" quotes'`
- Aspas duplas: `b"still allows embedded 'single' quotes"`.
- Aspas triplas: `b'''3 single quotes'''`, `b"""3 double quotes"""`

Apenas caracteres ASCII são permitidos em literais de bytes (independentemente do encoding declarado). Qualquer valor binário superior a 127 deverá ser inserido em bytes literais usando o Escape Sequence apropriada.

Assim como string literais, bytes literais também podem usar um prefixo `r` para desabilitar o processamento de sequências de escape. Veja strings para mais sobre as várias formas de bytes literais, incluindo suporte a sequências de escape.

Enquanto bytes literais e representações são baseados em texto ASCII, objetos bytes na verdade se comportam como sequências imutáveis de inteiros, com cada valor na sequência restrito aos limites `0 <= x < 256` (tentativas de violar essa restrição irão disparar *ValueError*). Isto é feito deliberadamente para enfatizar que enquanto muitos formatos binários incluem elementos baseados em ASCII e podem ser utilmente manipulados com alguns algoritmos orientados a texto, esse não é geralmente o caso para dados binários arbitrários (aplicar algoritmos de processamento de texto cegamente em formatos de dados binários que não são compatíveis com ASCII irá usualmente resultar em dados corrompidos).

Além das formas literais, os objetos bytes podem ser criados de várias outras maneiras:

- Um bytes preenchido com objetos zero com um comprimento especificado: `bytes(10)`

- De um iterável de inteiros: `bytes(range(20))`
- Copiando dados binários existentes através do protocolo de Buffer: `bytes(obj)`

Veja também os bult-ins *bytes*.

Como 2 dígitos hexadecimais correspondem precisamente a apenas um byte, números hexadecimais são um formato comumente usado para descrever dados binários. Portanto, o tipo bytes tem um método de classe adicional para ler dados nesse formato:

classmethod fromhex(*string*)
Este método de classe da classe *bytes* retorna um objeto bytes, decodificando o objeto string fornecido. A string deve conter dois dígitos hexadecimais por byte, com espaço em branco ASCII sendo ignorado.

```
>>> bytes.fromhex('2Ef0 F1f2  ')
b'.\xf0\xf1\xf2'
```

Alterado na versão 3.7: *bytes.fromhex()* agora ignora todos os espaços em branco em ASCII na string, não apenas espaços.

Uma função de conversão reversa existe para transformar um objeto bytes na sua representação hexadecimal.

hex()
Retorna um objeto string contendo dois dígitos hexadecimais para cada byte na instância.

```
>>> b'\xf0\xf1\xf2'.hex()
'f0f1f2'
```

Novo na versão 3.5.

Como objetos bytes são sequências de inteiros (certo parentesto a uma tupla), para um objeto bytes *b*, `b[0]` será um inteiro, enquanto `b[0:1]` será um objeto bytes de comprimento 1. (Isso contrasta com strings de texto, onde tanto o uso por índice quanto por fatiamento irão produzir uma string de comprimento 1)

A representação de objetos bytes utiliza o formato literal (`b'...'`), uma vez que ela é frequentemente mais útil do que, por exemplo, `bytes([46, 46, 46])`. Você sempre pode converter um objeto bytes em uma lista de inteiros usando `list(b)`.

Nota: Para usuários do Python 2.x: nas séries do Python 2.x, uma variedade de conversões implícitas entre strings de 8-bits (a coisa mais próxima a um tipo de dados binário embutido que o 2.x oferece) e Unicode strings era permitida. Isso era uma solução de contorno para compatibilidade retroativa para contabilizar o fato que o Python originalmente apenas suportava texto de 8-bits, e texto Unicode foi adicionado mais tarde. No Python 3.x, essas conversões implícitas se foram - conversões entre dados binários de 8-bits e texto Unicode devem ser explícitas, e bytes e objetos string irão sempre comparar como diferentes.

4.8.2 Bytearray Objects

Objetos *bytearray* são mutáveis, em contrapartida a objetos *bytes*.

class bytearray([*source*[, *encoding*[, *errors*]]])
Não existe sintaxe literal dedicada para objetos de bytearray, ao invés disso eles sempre são criados através da chamada do construtor:

- Criando uma instância vazia: `bytearray()`
- Criando uma instância cheia de zero com um determinado comprimento: `bytearray(10)`
- A partir de inteiros iteráveis: `bytearray(range(20))`

- Copiando dados binários existentes através do protocolo de buffer: `bytearray(b'Hi!')`

Como os objetos bytearray são mutáveis, os mesmos suportam as operações de seqüência *mutable* além das operações comuns de bytes e bytearray descritas em *Operações com Bytes e Bytearray*.

Veja também o tipo built-in *bytearray*.

Uma vez que 2 dígitos hexadecimais correspondem precisamente a um único byte, os números hexadecimais são um formato comumente usado para descrever dados binários. Consequentemente, o tipo de bytearray tem um método de classe adicional para ler dados nesse formato:

classmethod fromhex(*string*)

Este método de classe da classe *bytearray* retorna um objeto bytearray, decodificando o objeto string fornecido. A string deve conter dois dígitos hexadecimais por byte, com espaços em branco ASCII sendo ignorados.

```
>>> bytearray.fromhex('2Ef0 F1f2  ')
bytearray(b'.\xf0\xf1\xf2')
```

Alterado na versão 3.7: *bytearray.fromhex()* agora ignora todos os espaços em branco em ASCII na string, não apenas espaços.

Existe uma função de conversão reversa para transformar um objeto bytearray em sua representação hexadecimal.

hex()

Retorna um objeto string contendo dois dígitos hexadecimais para cada byte na instância.

```
>>> bytearray(b'\xf0\xf1\xf2').hex()
'f0f1f2'
```

Novo na versão 3.5.

Como os objetos bytearray são sequências de inteiros (semelhante a uma lista), para um objeto bytearray *b*, `b[0]` será um inteiro, enquanto que `b[0:1]` será um objeto bytearray de comprimento 1. (Isso contrasta com as Strings de texto, onde tanto a indexação como o fatiamento produzirão seqüências de comprimento 1)

A representação de objetos bytearray utiliza o formato literal bytes (`bytearray(b'...')`), uma vez que muitas vezes é mais útil do que, por exemplo, `bytearray([46, 46, 46])`. Você sempre pode converter um objeto bytearray em uma lista de inteiros usando `list(b)`.

4.8.3 Operações com Bytes e Bytearray

Ambos os tipos, bytes e os objetos bytearray suportam as operações de sequência common 1. Os mesmo interagem não apenas com operandos do mesmo tipo, mas com qualquer *bytes-like object*. Devido a esta flexibilidade, os mesmos podem ser misturados livremente em operações sem causar erros. No entanto, o tipo de retorno do resultado pode depender da ordem dos operandos.

Nota: Os métodos em bytes e objetos Bytearray não aceitam Strings como argumentos, assim como os métodos de Strings não aceitam bytes como argumentos. Por exemplo, devemos escrever:

```
a = "abc"
b = a.replace("a", "f")
```

e:

```
a = b"abc"
b = a.replace(b"a", b"f")
```

Alguns operações com bytes e bytearray assumem o uso de formatos binários compatíveis com ASCII e, portanto, devem ser evitados ao trabalhar com dados binários arbitrários. Essas restrições são abordadas a seguir.

Nota: O uso dessas operações baseadas em ASCII para manipular dados binários que não são armazenados num formato baseado em ASCII poderá resultar na corrupção de dados.

Os métodos a seguir em Bytes e Bytearray podem ser usados com dados binários arbitrários.

`bytes.`**`count`**(*sub*[, *start*[, *end*]])
`bytearray.`**`count`**(*sub*[, *start*[, *end*]])

Retorna o número de ocorrências não sobrepostas de subsequência *sub* no intervalo [*start*, *end*]. Os argumentos opcionais *start* e *end* são interpretados como na notação de fatiamento.

A subsequência a ser procurada poderá ser qualquer *bytes-like object* ou um inteiro no intervalo de 0 a 255.

Alterado na versão 3.3: Também aceita um número inteiro no intervalo de 0 a 255 como subsequência.

`bytes.`**`decode`**(*encoding="utf-8"*, *errors="strict"*)
`bytearray.`**`decode`**(*encoding="utf-8"*, *errors="strict"*)

Retorna uma String decodificada dos bytes fornecidos. A codificação padrão é `'utf-8'`. Os *erros* podem ser fornecidos para definir um esquema de tratamento de erros diferente. O padrão para *errors* é `'strict'`, o que significa que os erros de codificação geram uma exceção `UnicodeError`. Outros valores possíveis são `'ignore'`, `'replace'` e qualquer outro nome registrado pela função `codecs.register_error()`, veja a seção *Error Handlers*. Para obter uma lista dos possíveis encoding, veja a seção *Standard Encodings*.

Nota: Passando o argumento *encoding* para a classe `str` permite decodificar qualquer *bytes-like object* diretamente, sem precisar ter bytes temporário ou um objeto bytearray.

Alterado na versão 3.1: Adicionado suporte para argumentos keyword arguments.

`bytes.`**`endswith`**(*suffix*[, *start*[, *end*]])
`bytearray.`**`endswith`**(*suffix*[, *start*[, *end*]])

Retorna `True` se os dados binários encerra com o parâmetro *suffix* especificado, caso contrário retorna `False`. *suffix* também pode ser uma tupla de sufixos para averiguar. Com o parâmetro opcional *start*, a procura começa naquela posição. Com o parâmetro opcional *end*, o método encerra a comparação na posição fornecida.

O sufixo(es) para buscas pode ser qualquer termos *bytes-like object*.

`bytes.`**`find`**(*sub*[, *start*[, *end*]])
`bytearray.`**`find`**(*sub*[, *start*[, *end*]])

Retorna o índice mais baixo nos dados onde a sub-sequência *sub* é encontrada, tal que *sub* está contida na fatia `s[start:end]`. Argumentos opcionais *start* e *end* são interpretados como na notação de fatiamento. Retorna `-1` se *sub* não for localizada.

A subsequência a ser procurada poderá ser qualquer *bytes-like object* ou um inteiro no intervalo de 0 a 255.

Nota: O método `find()` deve ser usado apenas se você precisa saber a posição de *sub*. Para verificar se *sub* é uma substring ou não, use o operador `in`:

```
>>> b'Py' in b'Python'
True
```

Alterado na versão 3.3: Também aceita um número inteiro no intervalo de 0 a 255 como subsequência.

bytes.**index**(*sub*[, *start*[, *end*]])
bytearray.**index**(*sub*[, *start*[, *end*]])
Como `find()`, mas levanta a exceção `ValueError` quando a subsequência não for encontrada.

A subsequência a ser procurada poderá ser qualquer *bytes-like object* ou um inteiro no intervalo de 0 a 255.

Alterado na versão 3.3: Também aceita um número inteiro no intervalo de 0 a 255 como subsequência.

bytes.**join**(*iterable*)
bytearray.**join**(*iterable*)
Retorna um objeto bytes ou bytearray que é a concatenação das sequências de dados binários no *iterável*. Um `TypeError` será levantado se existirem quaisquer valores que não sejam *objeto byte ou similar*, no *iterável*, incluindo objetos `str`. O separador entre elementos é o conteúdo do objeto bytes ou bytearray que está fornecendo este método.

static bytes.**maketrans**(*from*, *to*)
static bytearray.**maketrans**(*from*, *to*)
Este método estático retorna uma tabela de tradução usável para `bytes.translate()`, que irá mapear cada caractere em *from* no caractere na mesma posição em *to*; *from* e *to* devem ambos ser *objeto byte ou similar* e ter o mesmo comprimento.

Novo na versão 3.1.

bytes.**partition**(*sep*)
bytearray.**partition**(*sep*)
Quebra a sequência na primeira ocorrência de *sep*, e retorna uma tupla com 3 elementos contendo a parte antes do separador, o próprio separator, e a parte após o separador. Se o separador não for encontrado, retorna uma tupla com 3 elementos contendo uma cópia da sequência original, seguido de dois bytes ou objetos bytearray vazios.

O separador para buscar pode ser qualquer termos *bytes-like object*.

bytes.**replace**(*old*, *new*[, *count*])
bytearray.**replace**(*old*, *new*[, *count*])
Retornar uma cópia da seqüência com todas as ocorrências de subsequências *antigas* substituídas por *novo*. Se o argumento opcional *count* for fornecido, apenas as primeiras ocorrências de *count* serão substituídas.

A subsequência para buscar e substituição pode ser qualquer termos *bytes-like object*.

Nota: A versão Bytearray deste método *não* opera no local - o mesmo sempre produz um novo objeto, mesmo que não tenha sido feitas alterações.

---

bytes.**rfind**(*sub*[, *start*[, *end*]])
bytearray.**rfind**(*sub*[, *start*[, *end*]])
Retorna o índice mais alto na seqüência onde a subsequência *sub* foi encontrada, de modo que *sub* esteja contido dentro de `s[start:end]`. Os argumentos opcionais *start* e *end* são interpretados como na notação de fatiamento. Caso ocorra algum problema será retorando `-1`.

A subsequência a ser procurada poderá ser qualquer *bytes-like object* ou um inteiro no intervalo de 0 a 255.

Alterado na versão 3.3: Também aceita um número inteiro no intervalo de 0 a 255 como subsequência.

bytes.**rindex**(*sub*[, *start*[, *end*]])
bytearray.**rindex**(*sub*[, *start*[, *end*]])
Semelhante a `rfind()` mas levanta `ValueError` quando a subsequência *sub* não é encontrada.

A subsequência a ser procurada poderá ser qualquer *bytes-like object* ou um inteiro no intervalo de 0 a 255.

Alterado na versão 3.3: Também aceita um número inteiro no intervalo de 0 a 255 como subsequência.

bytes.**rpartition**(*sep*)

bytearray.**rpartition**(*sep*)
Quebra a sequência na primeira ocorrência de *sep*, e retorna uma tupla com 3 elementos contendo a parte antes do separador, o próprio separator, e a parte após o separador. Se o separador não for encontrado, retorna uma tupla com 3 elementos contendo dois bytes ou objetos bytearray vazios, seguido por uma cópia da sequência original.

O separador para buscar pode ser qualquer termos *bytes-like object*.

bytes.**startswith**(*prefix*[, *start*[, *end*]])
bytearray.**startswith**(*prefix*[, *start*[, *end*]])
Retorna `True` se os dados binários começam com o *prefix* especificado, caso contrário retorna `False`. *prefix* também pode ser uma tupla de prefixos para procurar. Com o parâmetro opcional *start*, começa a procura na posição indicada. Com o parâmetro opcional *end*, encerra a procura na posição indicada.

Os prefix(os) para pesquisar podem ser qualquer *objeto-byte ou similar*.

bytes.**translate**(*table*, *delete=b''*)
bytearray.**translate**(*table*, *delete=b''*)
Retorna uma cópia dos bytes ou objeto bytearray, onde todas as ocorrências de bytes do argumento opcional *delete* são removidas, e os bytes restantes foram mapeados através da tabela de tradução fornecida, a qual deve ser um objeto bytes de comprimento 256.

Voce pode usar o método `bytes.maketrans()` para criar uma tabela de tradução.

Define o argumento *table* como `None` para traduções que excluem apenas caracteres:

```
>>> b'read this short text'.translate(None, b'aeiou')
b'rd ths shrt txt'
```

Alterado na versão 3.6: *delete* agora é suportado como um keyword argument.

Os seguintes métodos de objetos bytes e bytearray tem comportamentos padrões que assumem o uso de formatos binários compatíveis com ASCII, mas ainda podem ser usados com dados binários arbitrários através da passagem argumentos apropriados. Perceba que todos os métodos de bytearray nesta seção *não* alteram os argumentos, e ao invés disso produzem novos objetos.

bytes.**center**(*width*[, *fillbyte*])
bytearray.**center**(*width*[, *fillbyte*])
Retorna uma cópia do objeto centralizado em uma sequência de comprimento *width*. Preenchimento é feito usando o *fillbyte* especificado (o padrão é um espaço ASCII). Para objetos `bytes`, a sequência original é retornada se *width* é menor que ou igual a `len(s)`.

Nota: A versão Bytearray deste método *não* opera no local - o mesmo sempre produz um novo objeto, mesmo que não tenha sido feitas alterações.

---

bytes.**ljust**(*width*[, *fillbyte*])
bytearray.**ljust**(*width*[, *fillbyte*])
Retorna uma cópia do objeto alinhado a esquerda em uma sequência de comprimento *width*. Preenchimento é feito usando o *fillbyte* especificado (o padrão é um espaço ASCII). Para objetos `bytes`, a sequência original é retornada se *width* é menor que ou igual a `len(s)`.

Nota: A versão Bytearray deste método *não* opera no local - o mesmo sempre produz um novo objeto, mesmo que não tenha sido feitas alterações.

---

bytes.**lstrip**([*chars*])
bytearray.**lstrip**([*chars*])
Retorna uma cópia da sequência com bytes especificados no início e no final removidos. O argumento *chars* é uma

sequência binária que especifica o conjunto de bytes a serem removidos - o nome refere-se ao fato de este método é usualmente utilizado com caracteres ASCII. Se for omitido ou for `None`, o argumento *chars* irá remover por padrão os espaços em branco ASCII. O argumento *chars* não é um prefixo; ao contrário disso, todas as combinações dos seus seus valores são removidas:

```
>>> b'   spacious   '.lstrip()
b'spacious   '
>>> b'www.example.com'.lstrip(b'cmowz.')
b'example.com'
```

A sequência binária de valores de bytes a serem removidos pode ser qualquer *objeto byte ou similar*.

Nota: A versão Bytearray deste método *não* opera no local - o mesmo sempre produz um novo objeto, mesmo que não tenha sido feitas alterações.

`bytes.`**`rjust`**(*width*[, *fillbyte*])
`bytearray.`**`rjust`**(*width*[, *fillbyte*])

Retorna uma cópia do objeto alinhado a direita em uma sequência de comprimento *width*. Preenchimento é feito usando o *fillbyte* especificado (o padrão é um espaço ASCII). Para objetos *bytes*, a sequência original é retornada se *width* é menor que ou igual a `len(s)`.

Nota: A versão Bytearray deste método *não* opera no local - o mesmo sempre produz um novo objeto, mesmo que não tenha sido feitas alterações.

`bytes.`**`rsplit`**(*sep=None*, *maxsplit=-1*)
`bytearray.`**`rsplit`**(*sep=None*, *maxsplit=-1*)

Divide a sequência binária em subsequência do mesmo tipo, usando *sep* como um delimitador de string. Se *maxsplit* é fornecido, no máximo *maxsplit* divisões são feitas, aquelas que estão *mais à direita*. Se *sep* não é especificado ou é `None`, qualquer subsequência consistindo unicamente de espaço em branco ASCII é um separador. Exceto por dividir pela direita, *rsplit()* comporta-se como *split()*, o qual é descrito em detalhes abaixo.

`bytes.`**`rstrip`**([*chars*])
`bytearray.`**`rstrip`**([*chars*])

Retorna uma cópia da sequência com bytes especificados no final removidos. O argumento *chars* é uma sequência binária que especifica o conjunto de bytes a serem removidos - o nome refere-se ao fato de este método é usualmente utilizado com caracteres ASCII. Se for omitido ou for `None`, o argumento *chars* irá remover por padrão os espaços em branco ASCII. O argumento *chars* não é um sufixo; ao contrário disso, todas as combinações dos seus seus valores são removidas:

```
>>> b'   spacious   '.rstrip()
b'   spacious'
>>> b'mississippi'.rstrip(b'ipz')
b'mississ'
```

A sequência binária de valores de bytes a serem removidos pode ser qualquer *objeto byte ou similar*.

Nota: A versão Bytearray deste método *não* opera no local - o mesmo sempre produz um novo objeto, mesmo que não tenha sido feitas alterações.

`bytes.`**`split`**(*sep=None*, *maxsplit=-1*)
`bytearray.`**`split`**(*sep=None*, *maxsplit=-1*)

Divide a sequência binária em subsequências do mesmo tipo, usando *sep* como o delimitador de string. Se *maxsplit*

é fornecido e não-negativo, no máximo *maxsplit* divisões são feitas (logo, a lista terá no máximo `maxsplit+1` elementos). Se *maxsplit* não é especificado ou é `-1`, então não existe limite no número de divisões (todas as divisões possíveis são feitas).

Se *sep* é fornecido, delimitadores consecutivos não são agrupados juntos e eles são destinados para delimitar strings vazias (por exemplo `b'1,,2'.split(b',')` retorna `[b'1', b'', b'2']`). O argumento *sep* pode consistir de uma sequência de múltiplos bytes (por exemplo, `b'1<>2<>3'.split(b'<>')` retorna `[b'1', b'2', b'3']`). Separar uma sequência vazia com um separador especificado retorna `[b'']` ou `[bytearray(b'')]` dependendo do tipo do objeto que está sendo separado. O argumento *sep* pode ser qualquer *bytes-like object*.

Por exemplo:

```
>>> b'1,2,3'.split(b',')
[b'1', b'2', b'3']
>>> b'1,2,3'.split(b',', maxsplit=1)
[b'1', b'2,3']
>>> b'1,2,,3,'.split(b',')
[b'1', b'2', b'', b'3', b'']
```

Se *sep* não for especificado ou for `None`, um algoritmo diferente de separação é aplicado: ocorrências consecutivas de espaços em branco ASCII são consideradas como um separador único, e o resultado não conterá strings vazias no início ou no final, se a sequência tiver espaços em branco no início ou no final. Consequentemente, separar uma sequência vazia ou uma sequência que consiste apenas de espaços em branco ASCII sem um separador especificado, retorna `[]`.

Por exemplo:

```
>>> b'1 2 3'.split()
[b'1', b'2', b'3']
>>> b'1 2 3'.split(maxsplit=1)
[b'1', b'2 3']
>>> b'   1   2   3   '.split()
[b'1', b'2', b'3']
```

`bytes.`**`strip`**`(`[*chars*]`)`
`bytearray.`**`strip`**`(`[*chars*]`)`

Retorna uma cópia da sequência com os bytes especificados no início e no final removidos. O argumento *chars* é uma sequência binária que especifica o conjunto de bytes a serem removidos - o nome refere-se ao fato que este método é normalmente utilizado com caracteres ASCII. Se for omitido ou for `None`, o argumento *chars* irá remover por padrão os espaços em branco ASCII. O argumento *chars* não é um prefixo, nem um sufixo; ao contrário disso, todas as combinações dos seus seus valores são removidas:

```
>>> b'   spacious   '.strip()
b'spacious'
>>> b'www.example.com'.strip(b'cmowz.')
b'example'
```

A sequência binária de valores de bytes a serem removidos pode ser qualquer *objeto byte ou similar*.

Nota: A versão Bytearray deste método *não* opera no local - o mesmo sempre produz um novo objeto, mesmo que não tenha sido feitas alterações.

O seguintes métodos de bytes e objetos bytearray assumem o uso de formatos binários compatíveis com ASCII e não devem ser aplicados a dados binários arbitrários. Perceba que todos os métodos de bytearray nesta seção *não* operam in place, e ao invés disso produzem novos objetos.

bytes.**capitalize**()
bytearray.**capitalize**()
Retorna uma cópia da sequência com cada byte interpretado como um caractere ASCII, e o primeiro byte em letra maiúscula e o resto em letras minúsculas. Valores de bytes que não são ASCII são passados adiante sem mudanças.

Nota: A versão Bytearray deste método *não* opera no local - o mesmo sempre produz um novo objeto, mesmo que não tenha sido feitas alterações.

---

bytes.**expandtabs**(*tabsize=8*)
bytearray.**expandtabs**(*tabsize=8*)
Retorna uma cópia da sequência onde todos os caracteres de tabulação ASCII são substituídos por um ou mais espaços ASCII, dependendo da coluna atual e do tamanho fornecido para a tabulação. Posições de tabulação ocorrem a cada *tabsize* bytes (o padrão é 8, dada as posições de tabulação nas colunas 0, 8, 16 e assim por diante). Para expandir a sequência, a coluna atual é definida como zero e a sequência é examinada byte por byte. Se o byte é um caractere ASCII de tabulação (`b'\t'`), um ou mais caracteres de espaço são inseridos no resultado até que a coluna atual seja igual a próxima posição de tabulação. (O caractere de tabulação em si não é copiado.) Se o byte atual é um caractere ASCII de nova linha (`b'\n'`) ou de retorno (`b'\r'`), ele é copiado e a coluna atual é resetada para zero. Qualquer outro byte é copiado sem ser modificado e a coluna atual é incrementada em uma unidade independentemente de como o byte é representado quanto é impresso:

```
>>> b'01\t012\t0123\t01234'.expandtabs()
b'01      012     0123    01234'
>>> b'01\t012\t0123\t01234'.expandtabs(4)
b'01  012 0123    01234'
```

Nota: A versão Bytearray deste método *não* opera no local - o mesmo sempre produz um novo objeto, mesmo que não tenha sido feitas alterações.

---

bytes.**isalnum**()
bytearray.**isalnum**()
Retorna `True` se todos os bytes na sequência são caracteres alfabéticos ASCII ou dígitos decimais ASCII e a sequência não é vazia, `False` caso contrário. Caracteres alfabéticos ASCII são aqueles valores de byte na sequência `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`. Dígitos decimais ASCII são aqueles valores de byte na sequência `b'0123456789'`.

Por exemplo:

```
>>> b'ABCabc1'.isalnum()
True
>>> b'ABC abc1'.isalnum()
False
```

bytes.**isalpha**()
bytearray.**isalpha**()
Retorna `True` se todos os bytes na sequência são caracteres alfabéticos ASCII e a sequência não é vazia, `False` caso contrário. Caracteres alfabéticos ASCII são aqueles cujo valor dos bytes estão na sequência `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Por exemplo:

```
>>> b'ABCabc'.isalpha()
True
>>> b'ABCabc1'.isalpha()
False
```

bytes.**isascii**()
bytearray.**isascii**()
Retorna `True` se a sequência é vazia ou todos os bytes na sequência são ASCII, `False` caso contrário. Bytes ASCII estão no intervalo 0-0x7F.

Novo na versão 3.7.

bytes.**isdigit**()
bytearray.**isdigit**()
Retorna `True` se todos os bytes na sequência são dígitos decimais ASCII e a sequência não é vazia, `False` caso contrário. Dígitos decimais ASCII são aqueles cujos valores dos bytes estão na sequência `b'0123456789'`.

Por exemplo:

```
>>> b'1234'.isdigit()
True
>>> b'1.23'.isdigit()
False
```

bytes.**islower**()
bytearray.**islower**()
Retorna `True` se existe pelo menos um caractere minúsculo ASCII na sequência e nenhum caractere maiúsculo ASCII, `False` caso contrário.

Por exemplo:

```
>>> b'hello world'.islower()
True
>>> b'Hello world'.islower()
False
```

Caracteres minúsculos ASCII são aqueles cujos valores de bytes estão na sequência `b'abcdefghijklmnopqrstuvwxyz'`. Caracteres maiúsculos ASCII são aqueles cujos valores de bytes estão na sequência `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

bytes.**isspace**()
bytearray.**isspace**()
Retorna `True` se todos os bytes na sequência são espaço em branco ASCII e a sequência não é vazia, `False` caso contrário. Caracteres de espaço em branco ASCII são aqueles cujos valores de bytes estão na sequência `b' \t\n\r\x0b\f'` (espaço, tabulação, nova linha, retorno do cursor, tabulação vertical, formulário de entrada).

bytes.**istitle**()
bytearray.**istitle**()
Retorna `True` se a sequência é titlecased ASCII e a sequência não é vazia, `False` caso contrário. Veja *bytes.title()* para mais detalhes sobre a definição de "titlecased".

Por exemplo:

```
>>> b'Hello World'.istitle()
True
>>> b'Hello world'.istitle()
False
```

bytes.**isupper**()
bytearray.**isupper**()
Retorna `True` se existe pelo menos um caractere maiúsculo alfabético ASCII na sequência e nenhum caractere minúsculo ASCII, `False` caso contrário.

Por exemplo:

```
>>> b'HELLO WORLD'.isupper()
True
>>> b'Hello world'.isupper()
False
```

Caracteres minúsculos ASCII são aqueles cujos valores de bytes estão na sequência `b'abcdefghijklmnopqrstuvwxyz'`. Caracteres maiúsculos ASCII são aqueles cujos valores de bytes estão na sequência `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

`bytes.lower()`
`bytearray.lower()`

Retorna uma cópia da sequência com todos os caracteres maiúsculos ASCII convertidos para os seus correspondentes caracteres minúsculos.

Por exemplo:

```
>>> b'Hello World'.lower()
b'hello world'
```

Caracteres minúsculos ASCII são aqueles cujos valores de bytes estão na sequência `b'abcdefghijklmnopqrstuvwxyz'`. Caracteres maiúsculos ASCII são aqueles cujos valores de bytes estão na sequência `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Nota: A versão Bytearray deste método *não* opera no local - o mesmo sempre produz um novo objeto, mesmo que não tenha sido feitas alterações.

`bytes.splitlines(`*keepends=False*`)`
`bytearray.splitlines(`*keepends=False*`)`

Retorna uma lista das linhas na sequência binária, quebrando nos limites ASCII das linhas. Este método usa a abordagem de *novas linhas universais* para separar as linhas. Quebras de linhas não são incluídas nas listas resultantes a não ser que *keepends* seja fornecido e verdadeiro.

Por exemplo:

```
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines()
[b'ab c', b'', b'de fg', b'kl']
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
[b'ab c\n', b'\n', b'de fg\r', b'kl\r\n']
```

Ao contrário de *split()*, quando uma string delimitadora *sep* é fornecida, este método retorna uma lista vazia para a string vazia, e uma quebra de linha terminal não resulta em uma linha extra:

```
>>> b"".split(b'\n'), b"Two lines\n".split(b'\n')
([b''], [b'Two lines', b''])
>>> b"".splitlines(), b"One line\n".splitlines()
([], [b'One line'])
```

`bytes.swapcase()`
`bytearray.swapcase()`

Retorna uma cópia da sequência com todos os caracteres minúsculos ASCII convertidos para caracteres maiúsculos correspondentes, e vice-versa.

Por exemplo:

```
>>> b'Hello World'.swapcase()
b'hELLO wORLD'
```

Caracteres minúsculos ASCII são aqueles cujos valores de bytes estão na sequência `b'abcdefghijklmnopqrstuvwxyz'`. Caracteres maiúsculos ASCII são aqueles cujos valores de bytes estão na sequência `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Ao contrário de *`str.swapcase()`*, é sempre fato que `bin.swapcase().swapcase() == bin` para as versões binárias. Conversões maiúsculas/minúsculas são simétricas em ASCII, apesar que isso não é geralmente verdade para pontos de codificação arbitrários Unicode.

Nota: A versão Bytearray deste método *não* opera no local - o mesmo sempre produz um novo objeto, mesmo que não tenha sido feitas alterações.

`bytes.`**`title`**`()`
`bytearray.`**`title`**`()`

Retorna uma versão titlecased da sequência binária, onde palavras iniciam com um caractere ASCII com letra maiúscula e os caracteres restantes são em letras minúsculas. Bytes quem não possuem diferença entre maiúscula/minúscula não são alterados.

Por exemplo:

```
>>> b'Hello world'.title()
b'Hello World'
```

Caracteres minúsculos ASCII são aqueles cujos valores de byte estão na sequência `b'abcdefghijklmnopqrstuvwxyz'`. Caracteres maiúsculos ASCII são aqueles cujos valores de byte estão na sequência“b’ABCDEFGHIJKLMNOPQRSTUVWXYZ’“. Todos os outros valores de bytes ignoram maiúsculas/minúsculas.

O algoritmo usa uma definição simples independente de idioma para uma palavra, como grupos de letras consecutivas. A definição funciona em muitos contextos, mas isso significa que apóstrofes em contradições e possessivos formam limites de palavras, os quais podem não ser o resultado desejado:

```
>>> b"they're bill's friends from the UK".title()
b"They're Bill'S Friends From The Uk"
```

Uma solução alternativa para os apóstrofes pode ser construída usando expressões regulares:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(rb"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0:1].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase(b"they're bill's friends.")
b"They're Bill's Friends."
```

Nota: A versão Bytearray deste método *não* opera no local - o mesmo sempre produz um novo objeto, mesmo que não tenha sido feitas alterações.

`bytes.`**`upper`**`()`
`bytearray.`**`upper`**`()`

Retorna uma cópia da sequência com todos os caracteres minúsculos ASCII convertidos para sua contraparte maiúscula correspondente.

Por exemplo:

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

Caracteres minúsculos ASCII são aqueles cujos valores de bytes estão na sequência `b'abcdefghijklmnopqrstuvwxyz'`. Caracteres maiúsculos ASCII são aqueles cujos valores de bytes estão na sequência `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Nota: A versão Bytearray deste método *não* opera no local - o mesmo sempre produz um novo objeto, mesmo que não tenha sido feitas alterações.

`bytes.`**`zfill`**`(`*width*`)`
`bytearray.`**`zfill`**`(`*width*`)`

Retorna uma cópia da sequência preenchida a esquerda com dígitos `b'0'` ASCII para produzir uma sequência se comprimento *width*. Um sinal de prefixo inicial (`b'+'`/`b'-'`) é controlado através da inserção de preenchimento *depois* do caractere de sinal, ao invés de antes. Para objetos *bytes*, a sequência original é retornada se *width* é menor que ou igual a `len(seq)`.

Por exemplo:

```
>>> b"42".zfill(5)
b'00042'
>>> b"-42".zfill(5)
b'-0042'
```

Nota: A versão Bytearray deste método *não* opera no local - o mesmo sempre produz um novo objeto, mesmo que não tenha sido feitas alterações.

4.8.4 Estilo de Formatação de bytes `printf`-style

Nota: As operações de formatação descritas aqui exibem uma variedade de peculiaridades que levam a uma série de erros comuns (como a falha na exibição de Tuplas e Dicionários corretamente). Se o valor que está sendo impresso puder ser uma tupla ou dicionário, envolva-o numa tupla.

Objetos Bytes (`bytes`/`bytearray`) possuem uma operação integrada exclusiva: o operador `%` (modulo). Isso também é conhecido como o bytes de *formatação* ou o operador de *interpolação*. Dado `format % values` (onde *format* é um objeto bytes), as especificações de conversão `%` em *format* são substituídas por zero ou mais elementos de *values*. O efeito é semelhante ao uso da função `sprintf()` na linguagem C.

Se *format* requer um único argumento, *values* poderá ser um único objeto não-tuple.[5] Caso contrário, *values* deverá ser uma tupla com exatamente o número de itens especificados pelo objeto de formatação de Bytes, ou um único objeto de mapeamento (por exemplo, um dicionário).

Um especificador de conversão contém dois ou mais caracteres e tem os seguintes componentes, que devem aparecer nesta ordem:

1. O caractere `'%'`, que determina o início do especificador.
2. Mapeamento de Chaves (opcional), consistindo de uma sequência entre parênteses de caracteres (por exemplo, `(algumnome)`).
3. Flags de conversão (opcional), que afetam o resultado de alguns tipos de conversão.

4. Largura mínima do Campo(opcional). Se for especificado por `'*'` (asterisco), a largura real será lida a partir do próximo elemento da tupla em *values* e o objeto a converter virá após a largura mínima do campo e a precisão que é opcional.
5. Precisão (opcional), fornecido como uma `'.'` (ponto) seguido pela precisação. Se determinado como um `'*'` (um asterisco), a precisão real será lida a partir do próximo elemento da tupla em *values*, e o valor a converter virá após a precisão.
6. Modificador de Comprimento(opcional).
7. Tipos de Conversão

Quando o argumento a direita é um dicionário (ou outro tipo de mapeamento), então o formato no objeto bytes *deve* incluir um mapeamento de chaves entre parêntesis neste dicionário inserido imediatamente após o caractere `'%'`. O mapeamento de chaves seleciona o valor a ser formatado a partir do mapeamento. Por exemplo:

```
>>> print(b'%(language)s has %(number)03d quote types.' %
...       {b'language': b"Python", b"number": 2})
b'Python has 002 quote types.'
```

Nesse caso, nenhum especificador `*` poderá ocorrer num formato (uma vez que eles exigem uma lista de parâmetros sequenciais).

Os caracteres flags de conversão são:

Flag	Significado
`'#'`	A conversão de valor usará o "formulário alternativo" (em que definimos abaixo).
`'0'`	A conversão será preenchida por zeros para valores numéricos.
`'-'`	O valor convertido será ajustado à esquerda (substitui a conversão `'0'` se ambos forem fornecidos).
`' '`	(um espaço) Um espaço em branco deverá ser deixado antes de um número positivo (ou uma String vazia) produzido por uma conversão assinada.
`'+'`	Um sinal de caractere (`'+'` ou `'-'`) precedera a conversão (substituindo o sinalizador "space").

Um modificador de comprimento (`h`, `l`, ou `L`) pode estar presente, mas será ignorado, pois o mesmo não é necessário para o Python – então por exemplo `%ld` é idêntico a `%d`.

Os tipos de conversão são:

Conversão	Significado	Notas
`'d'`	Número decimal inteiro sinalizador.	
`'i'`	Número decimal inteiro sinalizador.	
`'o'`	Valor octal sinalizador.	(1)
`'u'`	Tipo obsoleto - é idêntico a `'d'`.	(8)
`'x'`	Sinalizador hexadecimal (minúsculas).	(2)
`'X'`	Sinalizador hexadecimal (maiúscula).	(2)
`'e'`	Formato exponencial de ponto flutuante (minúsculas).	(3)
`'E'`	Formato exponencial de ponto flutuante (maiúscula).	(3)
`'f'`	Formato decimal de ponto flutuante.	(3)
`'F'`	Formato decimal de ponto flutuante.	(3)
`'g'`	O formato de ponto flutuante. Usa o formato exponencial em minúsculas se o expoente for inferior a -4 ou não inferior a precisão, formato decimal, caso contrário.	(4)
`'G'`	Formato de ponto flutuante. Usa o formato exponencial em maiúsculas se o expoente for inferior a -4 ou não inferior que a precisão, formato decimal, caso contrário.	(4)
`'c'`	Byte simples (aceita objetos inteiros ou de byte único).	
`'b'`	Bytes (qualquer objeto que segue o buffer protocol o que possui `__bytes__()`).	(5)
`'s'`	`'s'` é um alias para `'b'` e só deve ser usado para bases de código Python2/3.	(6)
`'a'`	Bytes (converte qualquer objeto Python usando `repr(obj).encode('ascii', 'backslashreplace)`).	(5)
`'r'`	`'r'` é um alias para `'a'` e só deve ser usado para bases de código Python2/3.	(7)
`'%'`	Nenhum argumento é convertido, resultando um caractere `''%'` no resultado.	

Notas:

(1) A forma alternativa faz com que um especificador octal principal (`'0o'`) seja inserido antes do primeiro dígito.

(2) O formato alternativo produz um `'0x'` ou `'0X'` (dependendo se o formato `'x'` or `'X'` foi usado) para ser inserido antes do primeiro dígito.

(3) A forma alternativa faz com que o resultado sempre contenha um ponto decimal, mesmo que nenhum dígito o siga.

A precisão determina o número de dígitos após o ponto decimal e o padrão é 6.

(4) A forma alternativa faz com que o resultado sempre contenha um ponto decimal e os zeros à direita não sejam removidos, como de outra forma seriam.

A precisão determina o número de dígitos significativos antes e depois do ponto decimal e o padrão é 6.

(5) Se a precisão for `N`, a saída será truncada em caracteres `N`.

(6) `b'%s'` está obsoleto, mas não será removido durante a versão 3.x.

(7) `b'%r'` entrou em desuso, mas não serão removidos na versão 3.x.

(8) Veja **PEP 237**.

Nota: A versão Bytearray deste método *não* opera no local - o mesmo sempre produz um novo objeto, mesmo que não tenha sido feitas alterações.

Ver também:

PEP 461 - Adicionar formatação % para to bytes e bytearray

Novo na versão 3.5.

4.8.5 Memory Views

O objeto *memoryview* permite que o código Python acesse os dados internos de um objeto que suporte o :ref: 'buffer protocol 1' sem copia-lo.

class memoryview(*obj*)

Cria uma *memoryview* que referencia *obj*. *obj* deve suportar o protocolo de buffer. Objetos embutidos que suportam o protocolo de buffer incluem *bytes* e *bytearray*.

Uma *memoryview* tem a noção de um *elemento*, o qual é a unidade de memória atômica manipulada pelo objeto de origem *obj*. Para muitos tipos simples tais como *bytes* e *bytearray*, um elemento é um byte único, mas outros tipos tais como *array.array* podem ter elementos maiores.

`len(view)` é igual ao comprimento de *tolist*. Se `view.ndim = 0`, o comprimento é 1. Se `view.ndim = 1`, o comprimento é igual ao número de elementos na view. Para dimensões maiores, o comprimento é igual ao comprimento da representação de lista aninhada da view. O atributo *itemsize* irá lhe dar o número de bytes em um elemento individual.

Um *memoryview* suporta fatiamento e indexação para expor seus dados. Fatiamento unidimensional irá resultar em uma subview:

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

Se *format* é um dos especificadores de formatação nativos do módulo *struct*, indexar com um inteiro ou uma tupla de inteiros também é suportado, e retorna um *element* único com o tipo correto. Memoryviews unidimensionais podem ser indexadas com um inteiro ou uma tupla contendo um inteiro. Memoryviews multi-dimensionais podem ser indexadas com tuplas de exatamente *ndim* inteiros, onde *ndim* é o número de dimensões. Memoryviews zero-dimensionais podem ser indexadas com uma tupla vazia.

Aqui temos um exemplo usando um formato não-byte:

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333, 44444444])
>>> m = memoryview(a)
>>> m[0]
-11111111
>>> m[-1]
44444444
>>> m[::2].tolist()
[-11111111, -33333333]
```

Se o objeto subjacente é gravável, a memoryview suporta atribuição de fatias unidimensionais. Redimensionamento não é permitido:

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
```

(continua na próxima página)

(continuação da página anterior)

```
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
>>> data
bytearray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different structures
>>> v[2:6] = b'spam'
>>> data
bytearray(b'z1spam')
```

Memoryviews unidimensionais de tipos hasheáveis (apenas leitura) com formatos 'B', 'b' ou 'c' também são hasheáveis. O hash é definido como `hash(m) == hash(m.tobytes())`:

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::-2]) == hash(b'abcefg'[::-2])
True
```

Alterado na versão 3.3: Memoryviews unidimensionais agora podem ser fatiadas. Memoryviews unidimensionais com formatos 'B', 'b' ou 'c' agora são hasheáveis.

Alterado na versão 3.4: O memoryview agora é registrado automaticamente como uma classe *collections.abc.Sequence*

Alterado na versão 3.5: Atualmente, os memoryvies podem ser indexadas com uma tupla de números inteiros.

memoryview possui vários métodos:

__eq__(*exporter*)

Uma memoryview e um exportador **PEP 3118** são iguais se as suas formas são equivalentes e se todos os valores correspondentes são iguais quando os códigos de formatação dos respectivos operadores são interpretados usando a sintaxe *struct*.

Para o subconjunto *struct* a formatação de Strings atualmente suportadas por *tolist()*, `v` e `w` são iguais se `v.tolist() == w.tolist()`:

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

Se qualquer string de formatação não for suportada pelo módulo *struct*, então os objetos irão sempre

comparar como diferentes (mesmo se as strings de formatação e o conteúdo do buffer são idênticos):

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

Perceba que, assim como com números de ponto flutuante, `v is w` *não* implica em `v == w` para objetos memoryview.

Alterado na versão 3.3: Versões anteriores comparavam a memória bruta desconsiderando o formato do item e estrutura lógica do array.

tobytes()
Retorna os dados no buffer como um bytestring. Isso é equivalente a chamar o construtor de *bytes* na memoryview.

```
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
>>> bytes(m)
b'abc'
```

Para arrays não contíguos, o resultado é igual a representação de lista achatada com todos os elementos convertidos para bytes. *tobytes()* suporta todos os formatos de strings, incluindo aqueles que não estão na sintaxe do módulo *struct*.

hex()
Retorna um objeto string contendo dois dígitos hexadecimais para cada byte no buffer.

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

Novo na versão 3.5.

tolist()
Retorna os dados no buffer como uma lista de elementos.

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

Alterado na versão 3.3: *tolist()* agora suporta todos os formatos nativos de caracteres únicos na sintaxe do módulo *struct*, assim como representações multi-dimensionais.

release()
Libera o buffer subjacente exposto pelo objeto memoryview. Muitos objetos aceitam ações especiais quando a

view é mantida com eles (por exemplo, um *bytearray* iria temporariamente proibir o redimensionamento); portanto, chamar release() é útil para remover essas restrições (e liberar quaisquer recursos pendurados) o mais breve possível.

Depois que este método foi chamado, qualquer operação posterior na visão levanta um *ValueError* (exceto *release()*, o qual pode ser chamado múltiplas vezes):

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

O protocolo de gerenciamento de contexto pode ser usado para efeitos similares, usando a instrução `with`:

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

Novo na versão 3.2.

cast (*format*[, *shape*])

Converte uma memoryview para um novo formato ou forma. *shape* por padrão é `[byte_length//new_itemsize]`, o que significa que a view resultante será unidimensional. O valor de retorno é uma nova memoryview, mas o buffer por si mesmo não é copiado. Conversões suportadas são 1D -> C-*contíguo* r C-contíguo -> 1D.

O formato de destino é restrito a um elemento em formato nativo na sintaxe *struct*. Um dos formatos deve ser um formato de byte ('B', 'b' ou 'c'). O comprimento de bytes do resultado deve ser o mesmo que o comprimento original.

Converte de 1D/long para 1D/unsigned bytes:

```
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```

Converte de 1D/unsigned bytes para 1D/char:

```
>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview: invalid value for format "B"
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

Converte de 1D/bytes para 3D/ints para 1D/signed char:

```
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
>>> z.nbytes
48
```

Converte 1D/unsigned long para 2D/unsigned long:

```
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

Novo na versão 3.3.

Alterado na versão 3.5: O formato fonte não é mais restrito ao converter para uma visão de byte.

Existem também diversos atributos somente leitura disponíveis:

obj

O objeto subjacente da memoryview:

```
>>> b  = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True
```

Novo na versão 3.3.

nbytes

`nbytes == product(shape) * itemsize == len(m.tobytes())`. Este é a quantidade de espaço em bytes que o array deve usar em uma representação contígua. Ela não é necessariamente igual a `len(m)`:

```
>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> m = memoryview(a)
>>> len(m)
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12
```

Arrays Multi-dimensional:

```
>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.5]]
>>> len(y)
3
>>> y.nbytes
96
```

Novo na versão 3.3.

readonly

Um bool que indica se a memória é somente leitura.

format

Uma string contendo o formato (no estilo do módulo *`struct`*) para cada elemento na visão. Uma memoryview pode ser criada a partir de exportadores com strings de formato arbitrário, mas alguns métodos (ex: *`tolist()`*) são restritos a formatos de elementos nativos.

Alterado na versão 3.3: formato `'B'` agora é tratado de acordo com a sintaxe do módulo struct. Isso significa que `memoryview(b'abc')[0] == b'abc'[0] == 97`.

itemsize

O tamanho em Bytes de cada elemento do memoryview:

```
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
```

(continua na próxima página)

(continuação da página anterior)

```
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```

ndim
Um número inteiro que indica quantas dimensões de uma matriz multidimensional a memória representa.

shape
Uma tupla de inteiros de comprimento *ndim* dando a forma da memória como uma matriz N-dimensional.

Alterado na versão 3.3: Uma tupla vazia ao invés de `None` quando ndim = 0.

strides
Uma tupla de inteiros de comprimento de *ndim* dando o tamanho em bytes para acessar cada elemento de cada dimensão da matriz.

Alterado na versão 3.3: Uma tupla vazia ao invés de `None` quando ndim = 0.

suboffsets
Usado internamente para estilos de Arrays PIL. O valor é apenas informativo.

c_contiguous
Um bool que indica se a memória é C-*contiguous*.

Novo na versão 3.3.

f_contiguous
Um bool que indica se a memória é Fortran *contiguous*.

Novo na versão 3.3.

contiguous
Um bool que indica se a memória é *contiguous*.

Novo na versão 3.3.

4.9 Tipo Set — `set`, `frozenset`

Um objeto *set* é uma coleção não ordenada de objetos *hasheáveis* distintos. Usos comuns incluem testes de associação, remover duplicatas de uma sequência, e computar operações matemáticas tais como interseção, união, diferença, e diferença simétrica. (Para outros tipos de containers veja os tipos embutidos *dict*, *list*, e a classe *tuple*, e o módulo *collections*.)

Assim como outras coleções, sets suportam `x in set`, `len(set)`, e `for x in set`. Sendo uma coleção não ordenada, sets não armazenam posição de elementos ou ordem de inserção. Portanto, sets não suportam indexação, slicing, ou outros comportamentos similares de sequências.

Existem atualmente dois tipos de set embutidos, *set* e *frozenset*. O tipo *set* é mutável — o conteúdo pode ser alterado usando métodos como `add()` e `remove()`. Como ele é mutável, ele não tem valor hash e não pode ser usado como chave de dicionário ou um elemento de um outro set. O tipo *frozenset* é imutável e *hasheável* — seu conteúdo não pode ser alterado depois de ter sido criado; ele pode então ser usado como chave de dicionário ou como um elemento de outro set.

Sets não vazios (que não sejam frozensets) podem ser criados posicionando uma lista de elementos separados por vírgula dentro de chaves, por exemplo: `{'jack', 'sjoerd'}`, além do construtor *set*.

Os construtores de ambas as classes funcionam da mesma forma:

class set([*iterable*])
class frozenset([*iterable*])

Retorna um novo objeto set ou frozenset, cujos elementos são obtidos a partir de um *iterable*. Os elementos de um set devem ser *hasheável*. Para representar sets de sets, os sets internos devem ser objetos `frozenset`. Se *iterable* não for especificado, um novo set vazio é retornado.

Instâncias de `set` e `frozenset` fornecem as seguintes operações:

len(s)
Retorna o número de elementos no set *s* (cardinalidade de *s*).

x in s
Testa se *x* pertence a *s*.

x not in s
Testa se *x* não pertence a *s*.

isdisjoint(*other*)
Retorna `True` se o set não tem elementos em comum com *other*. Sets são deslocados (disjoint) se e somente se a sua interseção é o conjunto vazio.

issubset(*other*)
set <= other
Testa se cada elemento do set está contido em *other*.

set < other
Testa se o set é um subconjunto próprio de *other*, isto é, `set <= other and set != other`.

issuperset(*other*)
set >= other
Testa se cada elemento em *other* está contido no set.

set > other
Testa se o set é um superconjunto próprio de *other*, isto é, `set >= other and set != other`.

union(**others*)
set | other | ...
Retorna um novo set com elementos do set e de todos que estão em others.

intersection(**others*)
set & other & ...
Retorna um novo set com elementos comuns ao set e todos que estão em others.

difference(**others*)
set - other - ...
Retorna um novo set com elementos no set que não estão em others.

symmetric_difference(*other*)
set ^ other
Retorna um novo set com elementos estejam ou no set ou em *other*, mas não em ambos.

copy()
Retorna uma cópia rasa do set.

Perceba que, as versões não-operador dos métodos `union()`, `intersection()`, `difference()`, e `symmetric_difference()`, `issubset()`, e `issuperset()` irão aceitar qualquer iterável como um argumento. Em contraste, suas contrapartes baseadas em operadores exigem que seus argumentos sejam conjuntos. Isso impede construções sucetíveis a erros como `set('abc') & 'cbs'` e favorece a forma mais legível `set('abc').intersection('cbs')`.

Tanto *set* quanto *frozenset* suportam comparar um set contra outro set. Dois sets são iguais se e somente se, cada elemento de cada set está contido no outro set (cada um é um subconjunto do outro). Um set é menor que outro set se e somente se, o primeiro set é um subconjunto próprio do segundo set (é um subconjunto, mas não é igual). Um set é maior que outro set se e somente se, o primeiro set é um superconjunto próprio do segundo set (é um superconjunto, mas não é igual).

Instâncias de *set* são comparadas a instâncias de *frozenset* baseados nos seus membros. Por exemplo, `set('abc') == frozenset('abc')` retorna `True` e assim como `set('abc') in set([frozenset('abc')])`.

O subset e comparações de igualdade não generalizam para a função de ordenamento total. Por exemplo, quaisquer dois sets deslocados não vazios, não são iguais e não são subsets um do outro, então *todos* os seguintes retornam `False`: `a<b`, `a==b`, ou `a>b`.

Como sets apenas definem ordenamento parcial (subset de relacionamentos), a saída do método *list.sort()* é indefinida para listas e sets.

Elementos set, assim como chaves de dicionário, devem ser *hasheáveis*.

Operações binárias que misturam instâncias de *set* com *frozenset* retornam o tipo do primeiro operando. Por exemplo: `frozenset('ab') | set('bc')` retorna uma instância de *frozenset*.

A seguinte tabela lista operações disponíveis para *set* que não se aplicam para instâncias imutáveis de *frozenset*:

update(**others*)
set |= other | ...
Atualiza o set, adicionando elementos de others.

intersection_update(**others*)
set &= other & ...
Atualiza o set, mantendo somente elementos encontrados nele e em others.

difference_update(**others*)
set -= other | ...
Atualiza o set, removendo elementos encontrados em others.

symmetric_difference_update(*other*)
set ^= other
Atualiza o set, mantendo somente elementos encontrados em qualquer set, mas não em ambos.

add(*elem*)
Adiciona o elemento *elem* ao set.

remove(*elem*)
Remove o elemento *elem* do set. Levanta *KeyError* se *elem* não estiver contido no set.

discard(*elem*)
Remove o elemento *elem* do set se ele estiver presente.

pop()
Remove e retorna um elemento arbitrário do set. Levanta *KeyError* se o set estiver vazio.

clear()
Remove todos os elementos do set.

Perceba, as versões sem operator dos métodos *update()*, *intersection_update()*, *difference_update()*, e *symmetric_difference_update()* irão aceitar qualquer iterável como um argumento.

Perceba, o argumento *elem* para os métodos `__contains__()`, *remove()*, e *discard()* pode ser um set. Para suportar pesquisas por um frozenset equivalente, um frozenset temporário é criado a partir de *elem*.

4.10 Tipo de Mapeamento — `dict`

Um objeto de *mapeamento* mapeia valores *hasheáveis* para objetos arbitrários. Mapeamentos são objetos mutáveis. Existe no momento apenas um tipo de mapeamento padrão, o *dicionário*. (Para outros containers, veja as classes embutidas `list`, `set`, e `tuple`, e o módulo `collections`.)

As chaves de um dicionário são *quase* valores arbitrários. Valores que não são *hasheáveis*, isto é, valores contendo listas, dicionários ou outros tipos mutáveis (que são comparados por valor ao invés de por identidade do objeto) não devem ser usados como chaves. Tipos numéricos usados para chaves obedecem as regras normais para comparação numérica: se dois números comparados são iguais (tal como `1` e `1.0`), então eles podem ser usados intercambiavelmente para indexar a mesma entrada no dicionário. (Perceba entretanto, que como computadores armazenam números de ponto flutuante como aproximações, usualmente não é uma boa ideia utilizá-los como chaves para dicionários.)

Dicionários podem ser criados posicionando uma lista de pares `key: value` separados por vírgula dentro de chaves, por exemplo: `{'jack': 4098, 'sjoerd': 4127}` ou `{4098: 'jack', 4127: 'sjoerd'}`, ou usando o construtor de `dict`.

class dict(***kwarg*)
class dict(*mapping*, ***kwarg*)
class dict(*iterable*, ***kwarg*)

Retorna um novo dicionário inicializado a partir de um argumento posicional opcional, e um set de argumentos nomeados possivelmente vazio.

Se nenhum argumento posicional é fornecido, um dicionário vazio é criado. Se um argumento posicional é fornecido e é um objeto de mapeamento, um dicionário é criado com os mesmos pares de chave-valor que o objeto de mapeamento. Caso contrário, o argumento posicional deve ser um objeto *iterável*. Cada item no iterável deve ser por si mesmo um iterável com exatamente dois objetos. O primeiro objeto de cada item torna-se a chave no novo dicionário, e o segundo objeto, o valor correspondente. Se a chave ocorrer mais do que uma vez, o último valor para aquela chave torna-se o valor correspondente no novo dicionário.

Se argumentos nomeados são fornecidos, os argumentos nomeados e seus valores são adicionados ao dicionário criado a partir do argumento posicional. Se uma chave sendo adicionada já está presente, o valor do argumento nomeado substitui o valor do argumento posicional.

Para ilustrar, os seguintes exemplos todos retornam um dicionário igual a `{"one": 1, "two": 2, "three": 3}`:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

Fornecer argumentos nomeados conforme no primeiro exemplo somente funciona para chaves que são identificadores válidos no Python. Caso contrário, quaisquer chaves válidas podem ser usadas.

Estas são as operações que dicionários suportam (e portanto, tipos de mapeamento personalizados devem suportar também):

list(d)
Retorna uma lista de todas as chaves usadas no dicionário *d*.

len(d)
Retorna o número de items no dicionário *d*.

d[key]
Retorna o item de *d* com a chave *key*. Levanta um `KeyError` se *key* não estiver no mapeamento.

Se uma subclasse de um dict define um método `__missing__()` e *key* não estiver presente, a operação `d[key]` chama aquele método com a chave *key* como argumento. A operação `d[key]` então retorna ou levanta o que é retornado ou levantado pela chamada de `__missing__(key)`. Nenhuma operação ou método invoca `__missing__()`. Se `__missing__()` não for definido, então *KeyError* é levantado. `__missing__()` deve ser um método; ele não pode ser uma variável de instância:

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

O exemplo acima mostra parte da implementação de *collections.Counter*. Um método `__missing__` diferente é usado para *collections.defaultdict*.

d[key] = value
Define `d[key]` para *value*.

del d[key]
Remove `d[key]` desde o *d*. Levanta uma exceção *KeyError* se *key* não estiver em map.

key in d
Retorna `True` se *d* tiver uma chave *key*, senão `False`.

key not in d
Equivalente a `not key in d`.

iter(d)
Retorna um iterador para as chaves do dicionário. Isso é um atalho para `iter(d.keys())`.

clear()
Remove todos os itens do dicionário.

copy()
Retorna uma cópia superficial do dicionário.

classmethod fromkeys(*iterable*[, *value*])
Cria um novo dicionário com chaves provenientes de *iterable* e valores definidos como *value*.

fromkeys() is a class method that returns a new dictionary. *value* defaults to `None`.

get(*key*[, *default*])
Retorna o valor para *key* se *key* está no dicionário, caso contrário *default*. Se *default* não é fornecido, será usado o valor padrão `None`, de tal forma que este método nunca levanta um *KeyError*.

items()
Retorna uma nova visão dos items do dicionário (pares de `(key, value)`). Veja a *documentação de objetos de visão de dicionário*.

keys()
Retorna uma nova visão das chaves do dicionário. Veja a *documentação de objetos de visão de dicionário*.

pop(*key*[, *default*])
Se *key* está no dicionário, remove a mesma e retorna o seu valor, caso contrário retorna *default*. Se *default* não foi fornecido e *key* não está no dicionário, um *KeyError* é levantado.

popitem()
Remove e retorna um par `(key, value)` do dicionário. Pares são retornados como uma pilha, ou seja em ordem LIFO (last-in, first-out).

popitem() é útil para destrutivamente iterar sobre um dicionário, algo comumente usado em algoritmos de set. Se o dicionário estiver vazio, chamar *popitem()* levanta um *KeyError*.

Alterado na versão 3.7: Ordem LIFO agora é garantida. Em versões anteriores, *popitem()* iria retornar um par chave/valor arbitrário.

setdefault(*key*[, *default*])
Se *key* está no dicionário, retorna o seu valor. Se não, insere *key* com o valor *default* e retorna *default*. *default* por padrão usa o valor `None`.

update([*other*])
Atualiza o dicionário com os pares chave/valor existente em *other*, sobrescrevendo chaves existentes. Retorna `None`.

update() aceita ou outro objeto dicionário, ou um iterável de pares de chave/valor (como tuplas ou outros iteráveis de comprimento dois). Se argumentos nomeados são especificados, o dicionário é então atualizado com esses pares de chave/valor: `d.update(red=1, blue=2)`.

values()
Retorna uma nova visão dos valores do dicionário. Veja a *documentação de objetos de visão de dicionário*.

Uma comparação de igualdade entre uma visão de `dict.values()` e outra, sempre irá retornar `False`. Isso também se aplica ao comparar `dict.values()` entre si:

```
>>> d = {'a': 1}
>>> d.values() == d.values()
False
```

Dicionários são iguais se e somente se eles os mesmos pares `(key, value)` (independente de ordem). Comparações de ordem ('<', '<=', '>=', '>') levantam *TypeError*.

Dicionários preservam a ordem de inserção. Perceba que atualizar a chave não afeta a ordem. Chaves adicionadas após a deleção são inseridas no final.

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

Alterado na versão 3.7: Ordem do dicionário é garantida conforme a ordem de inserção. Este comportamento era um detalhe de implementação do CPython a partir da versão 3.6.

Ver também:

types.MappingProxyType podem ser usados para criar uma visão somente-leitura de um *dict*.

4.10.1 Objetos de visão de dicionário

Os objetos retornados por *dict.keys()*, *dict.values()* e *dict.items()* são *objetos de visão*. Eles fornecem uma visão dinâmica das entradas do dicionário, o que significa que quando o dicionário muda, a visão reflete essas mudanças.

Visões de dicionários podem ser iteradas para apresentar seus respectivos dados, e suportar testes de existência:

len(dictview)
Retorna o número de entradas no dicionário.

iter(dictview)
Retorna um iterador sobre as chaves, valores ou items (representados como tuplas de `(key, value)`) no dicionário.

Chaves e valores são iterados em ordem de inserção. Isso permite a criação de pares `(value, key)` usando *zip()*: `pairs = zip(d.values(), d.keys())`. Outra maneira de criar a mesma lista é `pairs = [(v, k) for (k, v) in d.items()]`.

Iterar sobre visões enquanto adiciona ou deleta entradas no dicionário pode levantar um *RuntimeError* ou falhar a iteração sobre todas as entradas.

Alterado na versão 3.7: Ordem do dicionário é garantida como a ordem de inserção.

x in dictview
Retorna `True` se *x* está nas chaves, valores ou items do dicionário subjacente (no último caso, *x* deve ser uma tupla de `(key, value)`).

Visões chave são similar a conjunto, como suas entradas são únicas e hasheáveis. Se todos os valores são hasheáveis, de tal forma que os pares `(chave, valor)` são únicos e hasheáveis, então a visão dos items também é um similar a conjunto. (Visões de valores não são tratadas de como similar a conjunto, pois as entradas geralmente não são únicas.) Para visões similares a conjunto, todas as operações definidas para a classe base abstrata *collections.abc.Set* estão disponíveis (por exemplo, ==, <, ou ^).

Um exemplo da utilização da visualização de dicionário:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order (insertion order)
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['bacon', 'spam']

>>> # set operations
```

(continua na próxima página)

(continuação da página anterior)

```
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}
```

4.11 Tipos de Gerenciador de Contexto

A instrução `with` do Python suporta o conceito de um contexto em tempo de execução definido por um gerenciador de contexto. Isto é implementado usando um par de métodos, que permitem que classes definidas pelo usuário especifiquem um contexto em tempo de execução, o qual é inicializado antes da execução das instruções, e encerrado quando as instruções terminam:

`contextmanager.__enter__()`
: Entra no contexto em tempo de execução e retorna este objeto ou outro objeto relacionado ao contexto em tempo de execução. O valor retornado por este método é ligado ao identificador na cláusula `as` da instrução `with` usando este gerenciador de contexto.

  Um exmplo de gerenciador de contexto que retorna a si mesmo é um *objeto arquivo*. Objeto arquivos retornam a si mesmos do método __enter__() para permitir que *open()* seja usado como a expressão de contexto em uma instrução `with`.

  Um exemplo de gerenciador de contexto que retorna um objeto relacionado é aquele retornado por *decimal.localcontext()*. Esses gerenciadores definem o contexto decimal ativo para uma cópia do contexto decimal original, e então retornam a cópia. Isso permite que mudanças sejam feitas no contexto decimal atual, no corpo contido na instrução `with`, sem afetar o código fora da instrução `with`.

`contextmanager.__exit__`(*exc_type*, *exc_val*, *exc_tb*)
: Sai do contexto em tempo de execução e retorna um flag Booleano indicando se qualquer exceção que ocorreu deve ser suprimida. Se uma exceção ocorreu enquanto era executado o corpo da instrução `with`, os argumentos contém o tipo da exceção, valor e informação da pilha de execução. Caso contrário, os tres argumentos são `None`.

  Retornar um valor verdadeiro deste método fará com que a instrução `with` suprima a exceção e continue a execução com a instrução imediatamente após a instrução `with`. Caso contrário a exceção continuará propagando após este método ter encerrado sua execução. Exceções que ocorrerem durante a execução deste método irão substituir qualquer exceção que tenha ocorrido dentro do corpo da instrução `with`.

  A exceção passada nunca deve ser re-levantada explicitamente - ao invés disso, este método deve retornar um valor falso para indicar que o método completou sua execução com sucesso, e não quer suprimir a exceção levantada. Isso permite ao código do gerenciador de contexto facilmente detectar se um método *__exit__()* realmente falhou ou não.

Python define diversos genreciadores de contexto para suportar facilmente sincronização de threads, solicita fechamento de arquivos ou outros objectos, e manipulação simples do contexto ativo de aritmética decimal. Os tipos especificados não são tratados especialmente além da sua implementação do protocolo do gerenciador de contexto. Veja o módulo *contextlib* para alguns exemplos.

Os *geradores* do python e o decorador *contextlib.contextmanager* fornecem um modo conveniente de implementar estes protocolos. Se uma função geradora é decorada com o decorador *contextlib.contextmanager*, ela irá retornar um gerenciador de contexto que implementa os métodos `__enter__()` e `__exit__()` necessários, ao invés do iterador produzido por uma função geradora não decorada.

Perceba que não existe nenhum slot específico para qualquer um desses métodos no tipo structure para objetos Python na API do Python/C. Tipos de extensão que desejam definir estes métodos devem fornecê-los como um método acessível normal do Python. Comparado com a sobrecarga de configurar o contexto em tempo de execução, a sobrecarga na pesquisa de dicionário em uma única classe é negligenciável.

4.12 Outros tipos embutidos

O interpretador suporta diversos outros tipos de objetos. Maior parte desses, suporta apenas uma ou duas operações.

4.12.1 Módulos

A única operação especial em um módulo é o acesso a um atributo: `m.nome`, onde *m* é um módulo e *nome* acessa o nome defino na tabela de símbolos de *m*. Atributos de módulo podem receber atribuição. (Perceba que a instrução `import` não é, estritamente falando, uma operação em um objeto do módulo; `import foo` não requer que um objeto do módulo chamado *foo* exista, ao invés disso requer uma *definição* (externa) de um módulo chamado *foo* em algum lugar.)

Um atributo especial de cada módulo é *__dict__*. Este é o dicionário contendo a tabela de símbolos do modulo. Modificar este dicionário vai na verdade modificar a tabela de símbolos do módulo, mas atribuição direta para o atributo *__dict__* não é possível (você pode escrever `m.__dict__['a'] = 1`, o qual define `m.a` para ser `1`, mas você não consegue escrever `m.__dict__ = {}`). Modificar *__dict__* diretamente não é recomendado.

Módulos embutidos no interpretador são escritos desta forma: `<module 'sys' (built-in)>`. Se carregados a partir de um arquivo, eles são escritos como `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`.

4.12.2 Classes e Instâncias de Classes

Veja objects e class para estes.

4.12.3 Funções

Objetos de funções são criados através da definição de funções. A única operação que pode ser feita em um objeto de função é chamá-la: `func(lista-de-argumentos)`.

Existem na verdade duas possibilidades de objetos de função: funções embutidas e funções definidas pelo usuário. Ambas suportam a mesma operação (chamar a função), mas a implementação é diferente, portanto os diferentes tipos de objetos.

Veja a funçao function para maiores informações.

4.12.4 Métodos

Métodos são funções que são chamadas usando a notação de atributo. Existem duas opções: métodos embutidos (tal como `append()` em listas) e métodos de instância de classe. Métodos embutidos são descritos com os tipos que suportam eles.

Se você acessar um método (uma função definida em um espaço de nomes de uma classe) através de uma instância, você obtém um objeto especial: um objeto *bound method* (também chamado de *método de instância*). Quando chamado, ele irá adicionar o argumento `self` para a lista de argumentos. Bound methods tem dois atributos somente leitura especiais: `m.__self__` é o objeto no qual o método opera, e `m.__func__` é a função que implementa o método. Chamar `m(arg-1, arg-2, ..., arg-n)` é completamente equivalente a chamar `m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)`.

Assim como objetos função, objetos de métodos vinculados tem suporte para receber atributos arbitrários. Entretanto, como atributos de métodos na verdade são armazenados no objeto função subjacente (`meth.__func__`), definir atributos de métodos em métodos vinculados não é permitido. Tentar definir um atributo em um método resulta em um *AttributeError* sendo levantado. A fim de definir um atributo de método, você precisa definir explicitamente ele no objeto função subjacente:

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

Veja o tipo types para maiores informações.

4.12.5 Objetos de Código

Objetos de código são usados pela implementação para representar código Python executável "pseudo-compilado", tal como corpo de uma função. Eles são diferentes de objetos de função porque eles não contém uma referência para os seus ambientes de execução global. Objetos de código são retornados pela função embutida *compile()* e podem ser extraídos de objetos de função através do seu atributo `__code__`. Veja também o módulo *code*.

Um objeto de código pode ser executado ou avaliado passando-o (ao invés da string fonte) para as funções embutidas *exec()* ou *eval()*.

Veja o tipo types para maiores informações.

4.12.6 Objetos de tipo

Objetos de tipos representam os vários tipos de objetos. Um tipo de um objeto é acessado pela função embutida *type()*. Não existem operações especiais sobre tipos. O módulo padrão *types* define nomes para todos os tipos padrão embutidos.

Tipos são escritos como isto: `<class 'int'>`.

4.12.7 O objeto Null

Este objeto é retornado por funções que não retornam um valor explicitamente. Ele não suporta operações especiais. Existe exatamente um objeto null, chamado `None` (um nome embutido). `type(None)()` produz o mesmo valor.

Ele é escrito como `None`.

4.12.8 O Objeto Ellipsis

Este objeto é comumente usado através de fatiamento (veja slicings). Ela não suporta operações especiais. Existe exatamente um objeto ellipsis, nomeado *Ellipsis* (um nome embutido). `type(Ellipsis)()` produz novamente *Ellipsis*.

Está escrito com `Ellipsis` ou `....`

4.12.9 O Objeto NotImplemented

Este objeto é retornado a partir de comparações e operações binárias quando elas são solicitadas para operar em tipos nos quais eles não suportam. Veja comparisons para mais informações. Existe exatamente um objeto `NotImplemented`. `type(NotImplemented)()` produz a instância singleton.

Está escrito como `NotImplemented`.

4.12.10 Valores Booleanos

Valores Booleanos são os dois objetos constantes `False` e `True`. Eles são usados para representar valores verdadeiros (apesar que outros valores também podem ser considerados falso ou verdadeiro). Em contextos numéricos (por exemplo quando usados como argumentos para um operator aritmético), eles se comportam como os inteiros 0 e 1, respectivamente. A função embutida *`bool()`* pode ser usada para converter qualquer valor para um Boolean, se o valor puder ser interpretado como um valor verdadeiro (veja a seção *Teste do Valor Verdade* acima).

Eles são escritos como `False` e `True`, respectivamente.

4.12.11 Objetos Internos

Veja a hierarquia de tipos padrão para esta informação. Ela descreve objetos de stack frame, objetos de traceback, e fatias de objetos.

4.13 Atributos Especiais

A implementação adiciona alguns atributos especiais somente-leitura para diversos tipos de objetos, onde eles são relevantes. Alguns desses não são reportados pela função embutida *`dir()`*.

`object.`**`__dict__`**
: Um dicionário ou outro objeto de mapeamento usado para armazenar os atributos (graváveis) de um objeto.

`instance.`**`__class__`**
: A classe à qual pertence uma instância de classe.

`class.`**`__bases__`**
: A tupla de classes base de um objeto classe.

`definition.`**`__name__`**
: O nome da classe, função, método, descritor, ou instância geradora.

`definition.`**`__qualname__`**
: O *nome qualificado* da classe, função, método, descritor, ou instância geradora.

  Novo na versão 3.3.

`class.`**`__mro__`**
: Este atributo é uma tupla de classes que são consideradas ao procurar por classes bases durante resolução de métodos.

`class.`**`mro`**`()`
: Este método pode ser substituído por uma metaclasse para customizar a ordem de resolução de métodos para suas instâncias. Ele é chamado na instanciação da classe, e o seu resultado é armazenado em *`__mro__`*.

`class.`**`__subclasses__`**`()`
: Cada classe mantém uma lista de referências fracas para suas subclasses imediatas. Este método retorna uma lista de todas essas referências que ainda estão vivas. Exemplo:

```
>>> int.__subclasses__()
[<class 'bool'>]
```

4.14 Integer string conversion length limitation

CPython has a global limit for converting between *int* and *str* to mitigate denial of service attacks. This limit *only* applies to decimal or other non-power-of-two number bases. Hexadecimal, octal, and binary conversions are unlimited. The limit can be configured.

The *int* type in CPython is an arbitrary length number stored in binary form (commonly known as a "bignum"). There exists no algorithm that can convert a string to a binary integer or a binary integer to a string in linear time, *unless* the base is a power of 2. Even the best known algorithms for base 10 have sub-quadratic complexity. Converting a large value such as `int('1' * 500_000)` can take over a second on a fast CPU.

Limiting conversion size offers a practical way to avoid CVE-2020-10735.

The limit is applied to the number of digit characters in the input or output string when a non-linear conversion algorithm would be involved. Underscores and the sign are not counted towards the limit.

When an operation would exceed the limit, a *ValueError* is raised:

```
>>> import sys
>>> sys.set_int_max_str_digits(4300)  # Illustrative, this is the default.
>>> _ = int('2' * 5432)
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300) for integer string conversion: value has 5432␣
↪digits; use sys.set_int_max_str_digits() to increase the limit.
>>> i = int('2' * 4300)
>>> len(str(i))
4300
>>> i_squared = i*i
>>> len(str(i_squared))
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300) for integer string conversion: value has 8599␣
↪digits; use sys.set_int_max_str_digits() to increase the limit.
>>> len(hex(i_squared))
7144
>>> assert int(hex(i_squared), base=16) == i*i  # Hexadecimal is unlimited.
```

The default limit is 4300 digits as provided in *sys.int_info.default_max_str_digits*. The lowest limit that can be configured is 640 digits as provided in *sys.int_info.str_digits_check_threshold*.

Verification:

```
>>> import sys
>>> assert sys.int_info.default_max_str_digits == 4300, sys.int_info
>>> assert sys.int_info.str_digits_check_threshold == 640, sys.int_info
>>> msg = int('578966293710682886880994035146873798396722250538762761564'
...           '9252925514383915483333812743580549779436104706260696366600'
...           '571186405732').to_bytes(53, 'big')
...
```

Novo na versão 3.7.14.

4.14.1 Affected APIs

The limitation only applies to potentially slow conversions between *int* and *str* or *bytes*:

- `int(string)` with default base 10.
- `int(string, base)` for all bases that are not a power of 2.
- `str(integer)`.
- `repr(integer)`.
- any other string conversion to base 10, for example `f"{integer}"`, `"{}".format(integer)`, or `b"%d" % integer`.

The limitations do not apply to functions with a linear algorithm:

- `int(string, base)` with base 2, 4, 8, 16, or 32.
- *int.from_bytes()* and *int.to_bytes()*.
- *hex()*, *oct()*, *bin()*.
- *Minilinguagem de especificação de formato* for hex, octal, and binary numbers.
- *str* to *float*.
- *str* to *decimal.Decimal*.

4.14.2 Configuring the limit

Before Python starts up you can use an environment variable or an interpreter command line flag to configure the limit:

- `PYTHONINTMAXSTRDIGITS`, e.g. `PYTHONINTMAXSTRDIGITS=640 python3` to set the limit to 640 or `PYTHONINTMAXSTRDIGITS=0 python3` to disable the limitation.
- `-X int_max_str_digits`, e.g. `python3 -X int_max_str_digits=640`
- `sys.flags.int_max_str_digits` contains the value of `PYTHONINTMAXSTRDIGITS` or `-X int_max_str_digits`. If both the env var and the `-X` option are set, the `-X` option takes precedence. A value of *-1* indicates that both were unset, thus a value of `sys.int_info.default_max_str_digits` was used during initialization.

From code, you can inspect the current limit and set a new one using these *sys* APIs:

- *sys.get_int_max_str_digits()* and *sys.set_int_max_str_digits()* are a getter and setter for the interpreter-wide limit. Subinterpreters have their own limit.

Information about the default and minimum can be found in *sys.int_info*:

- *sys.int_info.default_max_str_digits* is the compiled-in default limit.
- *sys.int_info.str_digits_check_threshold* is the lowest accepted value for the limit (other than 0 which disables it).

Novo na versão 3.7.14.

Cuidado: Setting a low limit *can* lead to problems. While rare, code exists that contains integer constants in decimal in their source that exceed the minimum threshold. A consequence of setting the limit is that Python source code containing decimal integer literals longer than the limit will encounter an error during parsing, usually at startup time or import time or even at installation time - anytime an up to date `.pyc` does not already exist for the code. A workaround for source that contains such large constants is to convert them to `0x` hexadecimal form as it has no limit.

Test your application thoroughly if you use a low limit. Ensure your tests run with the limit set early via the environment or flag so that it applies during startup and even during any installation step that may invoke Python to precompile `.py` sources to `.pyc` files.

4.14.3 Recommended configuration

The default `sys.int_info.default_max_str_digits` is expected to be reasonable for most applications. If your application requires a different limit, set it from your main entry point using Python version agnostic code as these APIs were added in security patch releases in versions before 3.11.

Example:

```
>>> import sys
>>> if hasattr(sys, "set_int_max_str_digits"):
...     upper_bound = 68000
...     lower_bound = 4004
...     current_limit = sys.get_int_max_str_digits()
...     if current_limit == 0 or current_limit > upper_bound:
...         sys.set_int_max_str_digits(upper_bound)
...     elif current_limit < lower_bound:
...         sys.set_int_max_str_digits(lower_bound)
```

If you need to disable it entirely, set it to `0`.

CAPÍTULO 5

Exceções embutidas

No Python, todas as exceções devem ser instâncias de uma classe derivada de *BaseException*. Em uma instrução `try` com uma cláusula `except` que menciona uma classe específica, essa cláusula também lida com qualquer classe de exceção derivada dessa classe (mas não com as classes de exceção a partir das quais *ela* é derivada). Duas classes de exceção que não são relacionadas por subclasse nunca são equivalentes, mesmo que tenham o mesmo nome.

As exceções embutidas listadas abaixo podem ser geradas pelo interpretador ou pelas funções embutidas. Exceto onde mencionado, eles têm um "valor associado" indicando a causa detalhada do erro. Pode ser uma sequência ou uma tupla de vários itens de informação (por exemplo, um código de erro e uma sequência que explica o código). O valor associado geralmente é passado como argumentos para o construtor da classe de exceção.

O código do usuário pode gerar exceções embutidas. Isso pode ser usado para testar um manipulador de exceções ou para relatar uma condição de erro "exatamente como" a situação na qual o interpretador gera a mesma exceção; mas lembre-se de que nada impede o código do usuário de gerar um erro inadequado.

As classes de exceção internas podem ser usadas como subclasses para definir novas exceções; Os programadores são incentivados a derivar novas exceções da classe *Exception* ou de uma de suas subclasses, e não de *BaseException*. Mais informações sobre a definição de exceções estão disponíveis no Tutorial do Python em tut-userexceptions.

Ao gerar (ou levantar novamente) uma exceção em uma cláusula `except` ou :keyword:‘ finally‘, `__context__` é automaticamente definida como a última exceção capturada; se a nova exceção não for tratada, o traceback exibido eventualmente incluirá as exceções de origem e a exceção final.

Ao levantar uma nova exceção (em vez de usar um `raise` simples para aumentar novamente a exceção que está sendo tratada), o contexto implícito da exceção pode ser complementado com uma causa explícita usando `from` com `raise`

```
raise new_exc from original_exc
```

A expressão a seguir `from` deve ser uma exceção ou `None`. Ela será definida como `__cause__` na exceção levantada. A definição de `__cause__` também define implicitamente o atributo `__suppress_context__` como `True`, de modo que o uso de `raise new_exc from None` substitui efetivamente a exceção antiga pela nova para fins de exibição (por exemplo, convertendo *KeyError* para *AttributeError*), deixando a exceção antiga disponível em `__context__` para introspecção durante a depuração.

O código de exibição padrão do traceback mostra essas exceções encadeadas, além do traceback da própria exceção. Uma exceção explicitamente encadeada em `__cause__` sempre é mostrada quando presente. Uma exceção implicitamente

encadeada em __context__ é mostrada apenas se __cause__ for *None* e __suppress_context__ for falso.

Em qualquer um dos casos, a exceção em si sempre é mostrada após todas as exceções encadeadas, de modo que a linha final do traceback sempre mostre a última exceção que foi levantada.

5.1 Classes base

As seguintes exceções são usadas principalmente como classes base para outras exceções.

exception BaseException
A classe base para todas as exceções internas. Não é para ser herdada diretamente por classes definidas pelo usuário (para isso, use *Exception*). Se *str()* for chamado em uma instância desta classe, a representação do(s) argumento(s) para a instância será retornada ou a string vazia quando não houver argumentos.

args
A tupla de argumentos fornecidos ao construtor de exceções. Algumas exceções embutidas (como *OSError*) esperam um certo número de argumentos e atribuem um significado especial aos elementos dessa tupla, enquanto outras são normalmente chamadas apenas com uma única string que fornece uma mensagem de erro.

with_traceback(*tb*)
Esse método define *tb* como o novo retorno para a exceção e retorna o objeto de exceção. Geralmente é usado no código de manipulação de exceção como este:

```
try:
    ...
except SomeException:
    tb = sys.exc_info()[2]
    raise OtherException(...).with_traceback(tb)
```

exception Exception
Todas as exceções embutidas que não saem para o sistema são derivadas dessa classe. Todas as exceções definidas pelo usuário também devem ser derivadas dessa classe.

exception ArithmeticError
A classe base para as exceções embutidas levantadas para vários erros aritméticos: *OverflowError*, *ZeroDivisionError*, *FloatingPointError*.

exception BufferError
Levantado quando uma operação relacionada a buffer não puder ser realizada.

exception LookupError
A classe base para as exceções levantadas quando uma chave ou índice usado em um mapeamento ou sequência é inválido: *IndexError*, *KeyError*. Isso pode ser levantado diretamente por *codecs.lookup()*.

5.2 Exceções concretas

As seguintes exceções são as que geralmente são levantados.

exception AssertionError
Levantado quando uma instrução `assert` falha.

exception AttributeError
Levantado quando uma referência de atributo (consulte attribute-reference) ou atribuição falha. (Quando um objeto não oferece suporte a referências ou atribuições de atributos, *TypeError* é levantado.)

exception EOFError
Levantado quando a função *input()* atinge uma condição de fim de arquivo (EOF) sem ler nenhum dado. (Note: os métodos `io.IOBase.read()` e *io.IOBase.readline()* retornam uma string vazia quando pressionam o EOF.)

exception FloatingPointError
Não usado atualmente.

exception GeneratorExit
Levantado quando *gerador* ou *coroutine* está fechado; veja `generator.close()` e `coroutine.close()`. Herda diretamente de *BaseException* em vez de *Exception*, já que tecnicamente não é um erro.

exception ImportError
Levantado quando a instrução `import` tem problemas ao tentar carregar um módulo. Também é gerado quando o "from list" em `from ... import` tem um nome que não pode ser encontrado.

Os atributos `name` e `path` podem ser configurados usando argumentos somente de palavra-chave para o construtor. Quando configurados, eles representam o nome do módulo que foi tentado ser importado e o caminho para qualquer arquivo que acionou a exceção, respectivamente.

Alterado na versão 3.3: Adicionados os atributos `name` e `path`.

exception ModuleNotFoundError
Uma subclasse de *ImportError* que é levantada por `import` quando um módulo não pôde ser localizado. Também é levantada quando `None` é encontrado em *sys.modules*.

Novo na versão 3.6.

exception IndexError
Levantada quando um índice de alguma sequência está fora do intervalo. (Índices de fatia são truncados sileciosamente para cair num intervalo permitido; se um índice não for um inteiro, *TypeError* é levantada.)

exception KeyError
Levantada quando uma chave de mapeamento (dicionário) não é encontrada no conjunto de chaves existentes.

exception KeyboardInterrupt
Levantada quando um usuário aperta a tecla de interrupção (normalmente `Control-C` ou `Delete`). Durante a execução, uma checagem de interrupção é feita regularmente. A exceção herda de *BaseException* para que não seja capturada acidentalmente por códigos que tratam *Exception* e assim evita que o interpretador saia.

exception MemoryError
Levantada quando uma operação fica sem memória mas a situação ainda pode ser recuperada (excluindo alguns objetos). O valor associado é uma string que indica o tipo de operação (interna) que ficou sem memória. Observe que, por causa da arquitetura de gerenciamento de memória subjacente (função `malloc()` do C), o interpretador pode não ser capaz de se recuperar completamente da situação; no entanto, levanta uma exceção para que um traceback possa ser impresso, no caso de um outro programa ser a causa.

exception NameError
Levantada quando um nome local ou global não é encontrado. Isso se aplica apenas a nomes não qualificados. O valor associado é uma mensagem de erro que inclui o nome que não pode ser encontrado.

exception NotImplementedError
Essa exceção é derivada da *RuntimeError*. Em classes base, definidas pelo usuário, os métodos abstratos devem gerar essa exceção quando requerem que classes derivadas substituam o método, ou enquanto a classe está sendo desenvolvida, para indicar que a implementação real ainda precisa ser adicionada.

Nota: Não deve ser usada para indicar que um operador ou método não será mais suportado – nesse caso deixe o operador / método indefinido ou, se é uma subclasse, defina-o como *None*.

Nota: `NotImplementedError` e `NotImplemented` não são intercambiáveis, mesmo que tenham nomes e propósitos similares. Veja *`NotImplemented`* para detalhes e casos de uso.

exception OSError([*arg*])
exception OSError(*errno*, *strerror*[, *filename*[, *winerror*[, *filename2*]]])

Esta exceção é levantada quando uma função do sistema retorna um erro relacionado ao sistema, incluindo falhas do tipo I/O como "file not found" ou "disk full"(não para tipos de argumentos não permitidos ou outro erro acessório.)

A segunda forma do construtor definir os atributos correspondentes, descritos abaixo. Os atributos terão valor *`None`* se não forem especificads. Por compatibilidade com versões anteriores, se três argumentos são passados, o atributo *`args`* contêm somente uma tupla de 2 elementos, os dois primeiros argumentos do construtor.

O construtor geralmente retorna uma subclasse de *`OSError`*, como descrito abaixo em *OS exceptions* . A subclasse particular depende do valor final de *`errno`*. Este comportamento ocorre apenas durante a construção direta ou por meio de um apelido de *`OSError`*, e não é herdado na criação de subclasses.

errno
Um código de erro numérico da variável C `errno`.

winerror
No Windows, isso fornece o código de erro nativo do Windows. O atributo *`errno`* é então uma tradução aproximada, em termos POSIX, desse código de erro nativo.

No Windows, se o argumento de construtor *winerror* for um inteiro, o atributo *`errno`* é determinado a partir do código de erro do Windows e o argumento *errno* é ignorado. Em outras plataformas, o argumento *winerror* é ignorado e o atributo *`winerror`* não existe.

strerror
A mensagem de erro correspondente, conforme fornecida pelo sistema operacional. É formatada pelas funções C `perror()` no POSIX e `FormatMessage()` no Windows.

filename
filename2
Para exceções que envolvem um caminho do sistema de arquivos (como *`open()`* ou *`os.unlink()`*), *`filename`* é o nome do arquivo passado para a função. Para funções que envolvem dois caminhos de sistema de arquivos (como: func: *os.rename*), *`filename2`* corresponde ao segundo nome de arquivo passado para a função.

Alterado na versão 3.3: *`EnvironmentError`*, *`IOError`*, *`WindowsError`*, *`socket.error`*, *`select.error`* e `mmap.error` foram fundidos em *`OSError`*, e o construtor pode retornar uma subclasse.

Alterado na versão 3.4: O atributo *`filename`* agora é o nome do arquivo original passado para a função, ao invés do nome codificado ou decodificado da codificação do sistema de arquivos. Além disso, o argumento e o atributo de construtor *filename2* foi adicionado.

exception OverflowError
Levantada quando o resultado de uma operação aritmética é muito grande para ser representada. Isso não pode ocorrer para inteiros (que prefere levantar *`MemoryError`* a desistir). No entanto, por motivos históricos, OverflowError às vezes é levantada para inteiros que estão fora de um intervalo obrigatório. Devido à falta de padronização do tratamento de exceção de ponto flutuante em C, a maioria das operações de ponto flutuante não são verificadas.

exception RecursionError
Esta exceção é derivada de *`RuntimeError`*. É levantada quando o interpretador detecta que a profundidade máxima de recursão (veja *`sys.getrecursionlimit()`*) foi excedida.

Novo na versão 3.5: Anteriormente, uma *`RuntimeError`* simples era levantada.

exception ReferenceError
Esta exceção é levantada quando um intermediário de referência fraca, criado pela função *weakref.proxy()*, é usado para acessar um atributo do referente após ter sido coletado como lixo. Para mais informações sobre referências fracas, veja o módulo *weakref*.

exception RuntimeError
Levantada quando um erro é detectado e não se encaixa em nenhuma das outras categorias. O valor associado é uma string indicando o que precisamente deu errado.

exception StopIteration
Levantada pela função embutida *next()* e o método *__next__()* de um *iterador* para sinalizar que não há mais itens produzidos pelo iterador.

O objeto exceção tem um único atributo `value`, que é fornecido como um argumento ao construir a exceção, e o padrão é *None*.

Quando uma função *geradora* ou corrotina retorna, uma nova instância *StopIteration* é levantada, e o valor retornado pela função é usado como o parâmetro `value` para o construtor da exceção.

Se um código gerador direta ou indiretamente levantar *StopIteration*, ele é convertido em uma *RuntimeError* (mantendo o *StopIteration* como a nova causa da exceção).

Alterado na versão 3.3: Adicionado o atributo `value` e a capacidade das funções geradoras de usá-lo para retornar um valor.

Alterado na versão 3.5: Introduzida a transformação RuntimeError via `from __future__ import generator_stop`, consulte **PEP 479**.

Alterado na versão 3.7: Habilita **PEP 479** para todo o código por padrão: um erro *StopIteration* levantado em um gerador é transformado em uma *RuntimeError*.

exception StopAsyncIteration
Deve ser levantada pelo método `__anext__()` de um objeto *iterador assíncrono* para parar a iteração.

Novo na versão 3.5.

exception SyntaxError
Levatada quando o analisador encontra um erro de sintaxe. Isso pode ocorrer em uma instrução `import`, em uma chamada às funções embutidas *exec()* ou *eval()*, ou ao ler o script inicial ou entrada padrão (também interativamente).

Instâncias desta classe têm atributos `filename`, `lineno`, `offset` e `text` para facilitar o acesso aos detalhes. *str()* da instância de exceção retorna apenas a mensagem.

exception IndentationError
Classe base para erros de sintaxe relacionados a indentação incorreta. Esta é uma subclasse de *SyntaxError*.

exception TabError
Levantada quando o indentação contém um uso inconsistente de tabulações e espaços. Esta é uma subclasse de *IndentationError*.

exception SystemError
Levantada quando o interpretador encontra um erro interno, mas a situação não parece tão grave para fazer com que perca todas as esperanças. O valor associado é uma string que indica o que deu errado (em termos de baixo nível).

Você deve relatar isso ao autor ou mantenedor do seu interpretador Python. Certifique-se de relatar a versão do interpretador Python (`sys.version`; também é impresso no início de uma sessão Python interativa), a mensagem de erro exata (o valor associado da exceção) e se possível a fonte do programa que acionou o erro.

exception SystemExit
Esta exceção é levantada pela função *sys.exit()*. Ele herda de *BaseException* em vez de exc:*Exception*

para que não seja acidentalmente capturado pelo código que captura *Exception*. Isso permite que a exceção se propague corretamente e faça com que o interpretador saia. Quando não é tratado, o interpretador Python sai; nenhum traceback (situação da pilha de execução) é impresso. O construtor aceita o mesmo argumento opcional passado para *sys.exit()*. Se o valor for um inteiro, ele especifica o status de saída do sistema (passado para a função C `exit()`); se for `None`, o status de saída é zero; se tiver outro tipo (como uma string), o valor do objeto é exibido e o status de saída é um.

Uma chamada para *sys.exit()* é traduzida em uma exceção para que os tratadores de limpeza (cláusulas :keyword:finally' das instruções `try`) possam ser executados, e para que um depurador possa executar um script sem correr o risco de perder o controle. A função *os._exit()* pode ser usada se for absolutamente necessário sair imediatamente (por exemplo, no processo filho após uma chamada para *os.fork()*).

code
O status de saída ou mensagem de erro transmitida ao construtor. (O padrão é `None`.)

exception TypeError
Levantada quando uma operação ou função é aplicada a um objeto de tipo inadequado. O valor associado é uma string que fornece detalhes sobre a incompatibilidade de tipo.

Essa exceção pode ser levantada pelo código do usuário para indicar que uma tentativa de operação em um objeto não é suportada e não deveria ser. Se um objeto deve ter suporte a uma dada operação, mas ainda não forneceu uma implementação, *NotImplementedError* é a exceção apropriada a ser levantada.

Passar argumentos do tipo errado (por exemplo, passar uma *list* quando um *int* é esperado) deve resultar em uma *TypeError*, mas passar argumentos com o valor errado (por exemplo, um número fora limites esperados) deve resultar em uma *ValueError*.

exception UnboundLocalError
Levantada quando uma referência é feita a uma variável local em uma função ou método, mas nenhum valor foi vinculado a essa variável. Esta é uma subclasse de *NameError*.

exception UnicodeError
Levantada quando ocorre um erro de codificação ou decodificação relacionado ao Unicode. É uma subclasse de *ValueError*.

UnicodeError possui atributos que descrevem o erro de codificação ou decodificação. Por exemplo, `err.object[err.start:err.end]` fornece a entrada inválida específica na qual o codec falhou.

encoding
O nome da codificação que levantou o erro.

reason
Uma string que descreve o erro de codec específico.

object
O objeto que o codec estava tentando codificar ou decodificar.

start
O primeiro índice de dados inválidos em *object*.

end
O índice após os últimos dados inválidos em *object*.

exception UnicodeEncodeError
Levantada quando ocorre um erro relacionado ao Unicode durante a codificação. É uma subclasse de *UnicodeError*.

exception UnicodeDecodeError
Levantada quando ocorre um erro relacionado ao Unicode durante a decodificação. É uma subclasse de *UnicodeError*.

exception UnicodeTranslateError
Levantada quando ocorre um erro relacionado ao Unicode durante a tradução. É uma subclasse de *UnicodeError*.

exception ValueError
Levantada quando uma operação ou função recebe um argumento que tem o tipo certo, mas um valor inadequado, e a situação não é descrita por uma exceção mais precisa, como *IndexError*.

exception ZeroDivisionError
Levantada quando o segundo argumento de uma divisão ou operação de módulo é zero. O valor associado é uma string que indica o tipo dos operandos e a operação.

As seguintes exceções são mantidas para compatibilidade com versões anteriores; a partir do Python 3.3, eles são apelidos de *OSError*.

exception EnvironmentError

exception IOError

exception WindowsError
Disponível apenas no Windows.

5.2.1 Exceções de sistema operacional

As seguintes exceções são subclasses de *OSError*, elas são levantadas dependendo do código de erro do sistema.

exception BlockingIOError
Levantada quando uma operação bloquearia em um objeto (por exemplo, soquete) definido para operação sem bloqueio. Corresponde a `errno` `EAGAIN`, `EALREADY`, `EWOULDBLOCK` e `EINPROGRESS`.

Além daquelas de *OSError*, *BlockingIOError* pode ter mais um atributo:

characters_written
Um inteiro contendo o número de caracteres gravados no fluxo antes de ser bloqueado. Este atributo está disponível ao usar as classes de E/S em buffer do módulo *io*.

exception ChildProcessError
Levantada quando uma operação em um processo filho falha. Corresponde a `errno` `ECHILD`.

exception ConnectionError
Uma classe base para problemas relacionados à conexão.

Suas subclasses são *BrokenPipeError*, *ConnectionAbortedError*, *ConnectionRefusedError* e *ConnectionResetError*.

exception BrokenPipeError
Uma subclasse de *ConnectionError*, levantada ao tentar escrever em um encadeamento enquanto a outra extremidade foi fechada, ou tentar escrever em um soquete que foi desligado para escrita. Corresponde a `errno` `EPIPE` e `ESHUTDOWN`.

exception ConnectionAbortedError
Uma subclasse de *ConnectionError*, levantada quando uma tentativa de conexão é cancelada pelo par. Corresponde a `errno` `ECONNABORTED`.

exception ConnectionRefusedError
Uma subclasse de *ConnectionError*, levantada quando uma tentativa de conexão é recusada pelo par. Corresponde a `errno` `ECONNREFUSED`.

exception ConnectionResetError
Uma subclasse de *ConnectionError*, levantada quando a conexão é reiniciada pelo par. Corresponde a `errno` `ECONNRESET`.

exception FileExistsError
Levantada ao tentar criar um arquivo ou diretório que já existe. Corresponde a `errno` `EEXIST`.

exception FileNotFoundError
Levantada quando um arquivo ou diretório é solicitado, mas não existe. Corresponde a `errno` `ENOENT`.

exception InterruptedError
Levantada quando uma chamada do sistema é interrompida por um sinal de entrada. Corresponde a `errno` *EINTR*.

Alterado na versão 3.5: Python agora tenta novamente chamadas de sistema quando uma *syscall* é interrompida por um sinal, exceto se o tratador de sinal levanta uma exceção (veja **PEP 475** para a justificativa), em vez de levantar *InterruptedError*.

exception IsADirectoryError
Levantada quando uma operação de arquivo (como *os.remove()*) é solicitada em um diretório. Corresponde a `errno` `EISDIR`.

exception NotADirectoryError
Levantada quando uma operação de diretório (como *os.listdir()*) é solicitada em algo que não é um diretório. Corresponde a `errno` `ENOTDIR`.

exception PermissionError
Levantada ao tentar executar uma operação sem os direitos de acesso adequados – por exemplo, permissões do sistema de arquivos. Corresponde a `errno` `EACCES` e `EPERM`.

exception ProcessLookupError
Levantada quando um determinado processo não existe. Corresponde a `errno` `ESRCH`.

exception TimeoutError
Levantada quando uma função do sistema expirou no nível do sistema. Corresponde a `errno` `ETIMEDOUT`.

Novo na versão 3.3: Todas as subclasses de *OSError* acima foram adicionadas.

Ver também:

PEP 3151 - Reworking the OS and IO exception hierarchy

5.3 Avisos

As seguintes exceções são usadas como categorias de aviso; veja a documentação de *Categorias de avisos* para mais detalhes.

exception Warning
Classe base para categorias de aviso.

exception UserWarning
Classe base para avisos gerados pelo código do usuário.

exception DeprecationWarning
Classe base para avisos sobre recursos descontinuados quando esses avisos se destinam a outros desenvolvedores Python.

exception PendingDeprecationWarning
Classe base para avisos sobre recursos que foram descontinuados e devem ser descontinuados no futuro, mas não foram descontinuados ainda.

Esta classe raramente é usada para emitir um aviso sobre uma possível descontinuação futura, é incomum, e *DeprecationWarning* é preferível para descontinuações já ativas.

exception SyntaxWarning
Classe base para avisos sobre sintaxe duvidosa.

exception RuntimeWarning
Classe base para avisos sobre comportamento duvidoso de tempo de execução.

exception FutureWarning
Classe base para avisos sobre recursos descontinuados quando esses avisos se destinam a usuários finais de aplicações escritas em Python.

exception ImportWarning
Classe base para avisos sobre prováveis erros na importação de módulos.

exception UnicodeWarning
Classe base para avisos relacionados a Unicode.

exception BytesWarning
Classe base para avisos relacionados a *bytes* e *bytearray*.

exception ResourceWarning
Base class for warnings related to resource usage. Ignored by the default warning filters.

Novo na versão 3.2.

5.4 Hierarquia das exceções

A hierarquia de classes para exceções embutidas é:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
```

(continua na próxima página)

(continuação da página anterior)

```
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

CAPÍTULO 6

Serviços de Processamento de Texto

Os módulos descritos neste capítulo fornecem uma ampla variedade de operações de manipulação de string e outros serviços de processamento de texto.

O módulo `codecs` descrito em *Serviços de Dados Binários* também é altamente relevante para o processamento de texto. Além disso, consulte a documentação do tipo string do Python em *Tipo de Sequência de Texto — str*.

6.1 `string` — Operações comuns de strings

Código Fonte: Lib/string.py

Ver também:

Tipo de Sequência de Texto — str

Métodos de String

6.1.1 Constantes de strings

As constantes definidas neste módulo são:

`string.`**`ascii_letters`**
: A concatenação das constantes `ascii_lowercase` e `ascii_uppercase` descritas abaixo. Este valor não depende da localidade.

`string.`**`ascii_lowercase`**
: As letras minúsculas `'abcdefghijklmnopqrstuvwxyz'`. Este valor não depende da localidade e não mudará.

`string.`**`ascii_uppercase`**
: As letras maiúsculas `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. Este valor não depende da localidade e não mudará.

string.**digits**
A string '0123456789'.

string.**hexdigits**
A string '0123456789abcdefABCDEF'.

string.**octdigits**
A string '01234567'.

string.**punctuation**
String of ASCII characters which are considered punctuation characters in the C locale.

string.**printable**
String de caracteres ASCII que são considerados imprimíveis. Esta é uma combinação de *digits*, *ascii_letters*, *punctuation* e *whitespace*.

string.**whitespace**
Uma string contendo todos os caracteres ASCII que são considerados espaços em branco. Isso inclui espaço de caracteres, tabulação, avanço de linha, retorno, avanço de formulário e tabulação vertical.

6.1.2 Formatação personalizada de strings

A classe embutida de string fornece a capacidade de fazer substituições de variáveis complexas e formatação de valor por meio do método *format()* descrito na **PEP 3101**. A classe *Formatter* no módulo *string* permite que você crie e personalize seus próprios comportamentos de formatação de strings usando a mesma implementação que o método embutido *format()*.

class string.**Formatter**
A classe *Formatter* tem os seguintes métodos públicos:

format(*format_string*, **args*, ***kwargs*)
O método principal da API. Leva uma string de formato e um conjunto arbitrário de argumentos posicionais e nomeados. É apenas um invólucro que chama *vformat()*.

Alterado na versão 3.7: Um argumento de string de formato é agora *somente-posicional*.

vformat(*format_string*, *args*, *kwargs*)
Esta função realiza o trabalho real de formatação. Ele é exposto como uma função separada para casos onde você deseja passar um dicionário predefinido de argumentos, ao invés de desempacotar e empacotar o dicionário como argumentos individuais usando a sintaxe *args e **kwargs. *vformat()* faz o trabalho de quebrar a string de formato em dados de caracteres e campos de substituição. Ela chama os vários métodos descritos abaixo.

Além disso, o *Formatter* define uma série de métodos que devem ser substituídos por subclasses:

parse(*format_string*)
Percorre format_string e retorna um iterável de tuplas (*literal_text*, *field_name*, *format_spec*, *conversion*). Isso é usado por *vformat()* para quebrar a string em texto literal ou campos de substituição.

Os valores na tupla representam conceitualmente um intervalo de texto literal seguido por um único campo de substituição. Se não houver texto literal (o que pode acontecer se dois campos de substituição ocorrerem consecutivamente), então *literal_text* será uma string de comprimento zero. Se não houver campo de substituição, então os valores de *field_name*, *format_spec* e *conversion* serão None.

get_field(*field_name*, *args*, *kwargs*)
Dado *field_name* conforme retornado por *parse()* (veja acima), converte-o em um objeto a ser formatado. Retorna uma tupla (obj, used_key). A versão padrão aceita strings no formato definido na **PEP 3101**, como "0[name]" ou "label.title". *args* e *kwargs* são como passados para *vformat()*. O valor de retorno *used_key* tem o mesmo significado que o parâmetro *key* para *get_value()*.

get_value(*key*, *args*, *kwargs*)
Obtém um determinado valor de campo. O argumento *key* será um inteiro ou uma string. Se for um inteiro, ele representa o índice do argumento posicional em *args*; se for uma string, então representa um argumento nomeado em *kwargs*.

O parâmetro *args* é definido para a lista de argumentos posicionais para `vformat()`, e o parâmetro *kwargs* é definido para o dicionário de argumentos nomeados.

For compound field names, these functions are only called for the first component of the field name; Subsequent components are handled through normal attribute and indexing operations.

Então, por exemplo, a expressão de campo '0.name' faria com que `get_value()` fosse chamado com um argumento *key* de 0. O atributo `name` será pesquisado após `get_value()` retorna chamando a função embutida `getattr()`.

Se o índice ou palavra-chave se referir a um item que não existe, um `IndexError` ou `KeyError` deve ser levantada.

check_unused_args(*used_args*, *args*, *kwargs*)
Implementa a verificação de argumentos não usados, se desejar. Os argumentos para esta função são o conjunto de todas as chaves de argumento que foram realmente referidas na string de formato (inteiros para argumentos posicionais e strings para argumentos nomeados) e uma referência a *args* e *kwargs* que foi passada para vformat. O conjunto de argumentos não utilizados pode ser calculado a partir desses parâmetros. Presume-se que `check_unused_args()` levata uma exceção se a verificação falhar.

format_field(*value*, *format_spec*)
`format_field()` simplesmente chama o global embutido `format()`. O método é fornecido para que as subclasses possam substituí-lo.

convert_field(*value*, *conversion*)
Converte o valor (retornado por `get_field()`) dado um tipo de conversão (como na tupla retornada pelo método `parse()`). A versão padrão entende os tipos de conversão "s" (str), "r" (repr) e "a" (ascii).

6.1.3 Sintaxe das strings de formato

O método `str.format()` e a classe `Formatter` compartilham a mesma sintaxe para strings de formato (embora no caso de `Formatter`, as subclasses possam definir sua própria sintaxe de string de formato). A sintaxe é relacionada a literais de string formatadas, mas há diferenças.

As strings de formato contêm "campos de substituição" entre chaves `{}`. Tudo o que não estiver entre chaves é considerado texto literal, que é copiado inalterado para a saída. Se você precisar incluir um caractere de chave no texto literal, ele pode ser escapado duplicando: `{{` e `}}`.

A gramática para um campo de substituição é a seguinte:

```
replacement_field  ::=  "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name         ::=  arg_name ("." attribute_name | "[" element_index "]")*
arg_name           ::=  [identifier | digit+]
attribute_name     ::=  identifier
element_index      ::=  digit+ | index_string
index_string       ::=  <any source character except "]"> +
conversion         ::=  "r" | "s" | "a"
format_spec        ::=  <described in the next section>
```

Em termos menos formais, o campo de substituição pode começar com um *field_name* que especifica o objeto cujo valor deve ser formatado e inserido na saída em vez do campo de substituição. O *field_name* é opcionalmente seguido por um

campo *conversion*, que é precedido por um ponto de exclamação `'!'`, e um *format_spec*, que é precedido por dois pontos `':'`. Eles especificam um formato não padrão para o valor de substituição.

Veja também a seção *Minilinguagem de especificação de formato*.

O próprio *field_name* começa com um *arg_name* que é um número ou uma palavra-chave. Se for um número, ele se refere a um argumento posicional, e se for uma palavra-chave, ele se refere a um argumento nomeado. Se os arg_names numéricos em uma string de formato forem 0, 1, 2, … em sequência, eles podem ser todos omitidos (não apenas alguns) e os números 0, 1, 2, … serão inseridos automaticamente nessa ordem . Como *arg_name* não é delimitado por aspas, não é possível especificar chaves de dicionário arbitrárias (por exemplo, as strings `'10'` ou `':-]'`) dentro de uma string de formato. O *arg_name* pode ser seguido por qualquer número de expressões de índice ou atributo. Uma expressão da forma `'.name'` seleciona o atributo nomeado usando *getattr()*, enquanto uma expressão da forma `'[index]'` faz uma pesquisa de índice usando `__getitem__()`.

Alterado na versão 3.1: Os especificadores de argumento posicional podem ser omitidos para *str.format()*, de forma que `'{} {}'.format(a, b)` é equivalente a `'{0} {1}'.format(a, b)`.

Alterado na versão 3.4: Os especificadores de argumento posicional podem ser omitidos para *Formatter*.

Alguns exemplos simples de string de formato:

```
"First, thou shalt count to {0}"  # References first positional argument
"Bring me a {}"                   # Implicitly references the first positional␣
↪argument
"From {} to {}"                   # Same as "From {0} to {1}"
"My quest is {name}"              # References keyword argument 'name'
"Weight in tons {0.weight}"       # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"   # First element of keyword argument 'players'.
```

O campo *conversion* causa uma coerção de tipo antes da formatação. Normalmente, o trabalho de formatação de um valor é feito pelo método `__format__()` do próprio valor. No entanto, em alguns casos, é desejável forçar um tipo a ser formatado como uma string, substituindo sua própria definição de formatação. Ao converter o valor em uma string antes de chamar `__format__()`, a lógica de formatação normal é contornada.

Três sinalizadores de conversão são atualmente suportados: `'!s'`, que chama *str()* no valor; `'!r'`, que chama *repr()*; e `'!a'`, que chama *ascii()*.

Alguns exemplos:

```
"Harold's a clever {0!s}"         # Calls str() on the argument first
"Bring out the holy {name!r}"     # Calls repr() on the argument first
"More {!a}"                       # Calls ascii() on the argument first
```

O campo *format_spec* contém uma especificação de como o valor deve ser apresentado, incluindo detalhes como largura do campo, alinhamento, preenchimento, precisão decimal e assim por diante. Cada tipo de valor pode definir sua própria "minilinguagem de formatação" ou interpretação de *format_spec*.

A maioria dos tipos embutidos oferece suporte a uma minilinguagem de formatação comum, que é descrita na próxima seção.

Um campo *format_spec* também pode incluir campos de substituição aninhados dentro dele. Esses campos de substituição aninhados podem conter um nome de campo, sinalizador de conversão e especificação de formato, mas um aninhamento mais profundo não é permitido. Os campos de substituição em format_spec são substituídos antes que a string *format_spec* seja interpretada. Isso permite que a formatação de um valor seja especificada dinamicamente.

Veja a seção *Exemplos de formato* para alguns exemplos.

Minilinguagem de especificação de formato

"Especificações de formato" são usadas nos campos de substituição contidos em uma string de formato para definir como os valores individuais são apresentados (consulte *Sintaxe das strings de formato* e f-strings). Elas também podem ser passadas diretamente para a função embutida `format()`. Cada tipo formatável pode definir como a especificação do formato deve ser interpretada.

A maioria dos tipos embutidos implementa as seguintes opções para especificações de formato, embora algumas das opções de formatação sejam suportadas apenas pelos tipos numéricos.

Uma convenção geral é que uma especificação de formato vazia produz o mesmo resultado como se você tivesse chamado `str()` no valor. Uma especificação de formato não vazio normalmente modifica o resultado.

A forma geral de um *especificador de formato padrão* é:

```
format_spec      ::=  [[fill]align][sign][#][0][width][grouping_option][.precision][type]
fill             ::=  <any character>
align            ::=  "<" | ">" | "=" | "^"
sign             ::=  "+" | "-" | " "
width            ::=  digit+
grouping_option  ::=  "_" | ","
precision        ::=  digit+
type             ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s
```

Se um valor *align* válido for especificado, ele pode ser precedido por um caractere de preenchimento *fill* que pode ser qualquer caractere e o padrão é um espaço se omitido. Não é possível usar uma chave literal ("`{`" ou "`}`") como o caractere *fill* em uma string formatada literal ou ao usar o método `str.format()`. No entanto, é possível inserir uma chave com um campo de substituição aninhado. Esta limitação não afeta a função `format()`.

O significado das várias opções de alinhamento é o seguinte:

Opção	Significado
`'<'`	Força o alinhamento à esquerda do campo dentro do espaço disponível (este é o padrão para a maioria dos objetos).
`'>'`	Força o alinhamento à direita do campo dentro do espaço disponível (este é o padrão para números).
`'='`	Força o preenchimento a ser colocado após o sinal (se houver), mas antes dos dígitos. É usado para imprimir campos na forma "+000000120". Esta opção de alinhamento só é válida para tipos numéricos. Torna-se o padrão quando "0" precede imediatamente a largura do campo.
`'^'`	Força a centralização do campo no espaço disponível.

Observe que, a menos que uma largura de campo mínima seja definida, a largura do campo sempre será do mesmo tamanho que os dados para preenchê-lo, de modo que a opção de alinhamento não tem significado neste caso.

A opção *sign* só é válida para tipos numéricos e pode ser um dos seguintes:

Opção	Significado
`'+'`	indica que um sinal deve ser usado para números positivos e negativos.
`'-'`	indica que um sinal deve ser usado apenas para números negativos (este é o comportamento padrão).
espaço	indica que um espaço inicial deve ser usado em números positivos e um sinal de menos em números negativos.

The '#' option causes the "alternate form" to be used for the conversion. The alternate form is defined differently for different types. This option is only valid for integer, float, complex and Decimal types. For integers, when binary, octal, or hexadecimal output is used, this option adds the prefix respective '0b', '0o', or '0x' to the output value. For floats, complex and Decimal the alternate form causes the result of the conversion to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it. In addition, for 'g' and 'G' conversions, trailing zeros are not removed from the result.

A opção ',' sinaliza o uso de uma vírgula para um separador de milhares. Para um separador que reconhece a localidade, use o tipo de apresentação inteiro 'n'.

Alterado na versão 3.1: Adicionada a opção ',' (veja também **PEP 378**).

A opção '_' sinaliza o uso de um sublinhado para um separador de milhares para tipos de apresentação de ponto flutuante e para o tipo de apresentação de inteiro 'd'. Para os tipos de apresentação inteiros 'b', 'o', 'x' e 'X', sublinhados serão inseridos a cada 4 dígitos. Para outros tipos de apresentação, especificar esta opção é um erro.

Alterado na versão 3.6: Adicionada a opção '_' (veja também **PEP 515**).

width é um número inteiro decimal que define a largura total mínima do campo, incluindo quaisquer prefixos, separadores e outros caracteres de formatação. Se não for especificado, a largura do campo será determinada pelo conteúdo.

Quando nenhum alinhamento explícito é fornecido, preceder o campo *largura* com um caractere zero ('0') habilita o preenchimento por zero com reconhecimento de sinal para tipos numéricos. Isso é equivalente a um caractere de fill* de '0' com um tipo de *alignment* de '='.

precision é um número decimal que indica quantos dígitos devem ser exibidos depois do ponto decimal para um valor de ponto flutuante formatado com 'f' e 'F', ou antes e depois do ponto decimal para um valor de ponto flutuante formatado com 'g' ou 'G'. Para tipos não numéricos, o campo indica o tamanho máximo do campo – em outras palavras, quantos caracteres serão usados do conteúdo do campo. *precision* não é permitido para valores inteiros.

Finalmente, o *type* determina como os dados devem ser apresentados.

Os tipos de apresentação de string disponíveis são:

Tipo	Significado
's'	Formato de string. Este é o tipo padrão para strings e pode ser omitido.
None	O mesmo que 's'.

Os tipos de apresentação inteira disponíveis são:

Tipo	Significado
'b'	Formato binário. Exibe o número na base 2.
'c'	Caractere. Converte o inteiro no caractere Unicode correspondente antes de imprimir.
'd'	Decimal Integer. Outputs the number in base 10.
'o'	Octal format. Outputs the number in base 8.
'x'	Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9.
'X'	Formato hexadecimal. Produz o número na base 16, usando letras maiúsculas para os dígitos acima de 9.
'n'	Número. É o mesmo que 'd', exceto que usa a configuração local atual para inserir os caracteres separadores de número apropriados.
None	O mesmo que 'd'.

Além dos tipos de apresentação acima, os inteiros podem ser formatados com os tipos de apresentação de ponto flutuante listados abaixo (exceto 'n' e None). Ao fazer isso, *float()* é usado para converter o inteiro em um número de ponto flutuante antes da formatação.

The available presentation types for floating point and decimal values are:

Tipo	Significado
`'e'`	Exponent notation. Prints the number in scientific notation using the letter 'e' to indicate the exponent. The default precision is `6`.
`'E'`	Exponent notation. Same as `'e'` except it uses an upper case 'E' as the separator character.
`'f'`	Fixed-point notation. Displays the number as a fixed-point number. The default precision is `6`.
`'F'`	Notação de ponto fixo. O mesmo que `'f'`, mas converte `nan` para `NAN` e `inf` para `INF`.
`'g'`	General format. For a given precision `p >= 1`, this rounds the number to `p` significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude. The precise rules are as follows: suppose that the result formatted with presentation type `'e'` and precision `p-1` would have exponent `exp`. Then if `-4 <= exp < p`, the number is formatted with presentation type `'f'` and precision `p-1-exp`. Otherwise, the number is formatted with presentation type `'e'` and precision `p-1`. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it, unless the `'#'` option is used. Infinito positivo e negativo, zero positivo e negativo e nans, são formatados como `inf`, `-inf`, `0`, `-0` e `nan`, respectivamente, independentemente da precisão. A precision of `0` is treated as equivalent to a precision of `1`. The default precision is `6`.
`'G'`	Formato geral. O mesmo que `'g'`, exceto muda para `'E'` se o número ficar muito grande. As representações de infinito e NaN também são maiúsculas.
`'n'`	Número. É o mesmo que `'g'`, exceto que usa a configuração da localidade atual para inserir os caracteres separadores de número apropriados.
`'%'`	Porcentagem. Multiplica o número por 100 e exibe no formato fixo (`'f'`), seguido por um sinal de porcentagem.
None	Similar to `'g'`, except that fixed-point notation, when used, has at least one digit past the decimal point. The default precision is as high as needed to represent the particular value. The overall effect is to match the output of *`str()`* as altered by the other format modifiers.

Exemplos de formato

Esta seção contém exemplos da sintaxe de:meth:*str.format* e comparação com a antiga formatação `%`.

Na maioria dos casos a sintaxe é semelhante à antiga formatação de `%`, com a adição de `{}` e com `:` usado em vez de `%`. Por exemplo, `'%03.2f'` pode ser traduzido para `'{:03.2f}'`.

A nova sintaxe de formato também oferece suporte a opções novas e diferentes, mostradas nos exemplos a seguir.

Acessando os argumentos por posição:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 3.1+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')      # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # arguments' indices can be repeated
'abracadabra'
```

Acessando os argumentos por nome:

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.
↪81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

Acessando os atributos dos argumentos:

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.
↪0.'
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

Acessando os itens dos argumentos:

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
```

Substituindo `%s` e `%r`:

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

Alinhando o texto e especificando uma largura:

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'***********centered***********'
```

Substituindo `%+f`, `%-f` e `% f` e especificando um sinal:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # show a space for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # show only the minus -- same as '{:f}; {:f}'
'3.140000; -3.140000'
```

Substituindo `%x` e `%o` e convertendo o valor para bases diferentes:

```
>>> # format also supports binary numbers
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
```

Usando a vírgula como um separador de milhares:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

Expressando uma porcentagem:

```
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

Usando formatação específica do tipo:

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

Argumentos de aninhamento e exemplos mais complexos:

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>right'
>>>
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base, width=width), end=' ')
...     print()
...
    5     5     5   101
    6     6     6   110
    7     7     7   111
    8     8    10  1000
    9     9    11  1001
   10     A    12  1010
   11     B    13  1011
```

6.1.4 Strings de modelo

Strings de modelo fornecem substituições de string mais simples, conforme descrito em **PEP 292**. Um caso de uso primário para strings de modelo é para internacionalização (i18n), uma vez que, nesse contexto, a sintaxe e a funcionalidade mais simples tornam mais fácil traduzir do que outros recursos embutidos de formatação de strings no Python. Como um exemplo de biblioteca construída sobre strings de modelo para i18n, veja o pacote flufl.i18n.

Strings de modelo oferecem suporte a substituições baseadas em `$`, usando as seguintes regras:

- `$$` é um escape; é substituído por um único `$`.
- `$identifier` nomeia um espaço reservado de substituição correspondendo a uma chave de mapeamento de `"identifier"`. Por padrão, `"identifier"` é restrito a qualquer string ASCII alfanumérica que não faz distinção entre maiúsculas e minúsculas (incluindo sublinhados) que começa com um sublinhado ou letra ASCII. O primeiro caractere não identificador após o caractere `$` termina esta especificação de espaço reservado.
- `${identifier}` é equivalente a `$identifier`. É necessário quando caracteres identificadores válidos seguem o marcador de posição, mas não fazem parte do marcador, como `"${noun}ification"`.

Qualquer outra ocorrência de `$` na string resultará em uma *`ValueError`* sendo levantada.

O módulo *`string`* fornece uma classe *`Template`* que implementa essas regras. Os métodos de *`Template`* são:

class string.**Template**(*template*)
O construtor recebe um único argumento que é a string de modelo.

substitute(*mapping*, ***kwds*)
Executa a substituição do modelo, retornando uma nova string. *mapping* é qualquer objeto dicionário ou similar com chaves que correspondem aos marcadores de posição no modelo. Como alternativa, você pode fornecer argumentos nomeados, os quais são espaços reservados. Quando *mapping* e *kwds* são fornecidos e há duplicatas, os marcadores de *kwds* têm precedência.

safe_substitute(*mapping*, ***kwds*)
Como *`substitute()`*, exceto que se os espaços reservados estiverem faltando em *mapping* e *kwds*, em vez de levantar uma exceção *`KeyError`*, o espaço reservado original aparecerá na string resultante intacta. Além disso, ao contrário de *`substitute()`*, qualquer outra ocorrência de `$` simplesmente retornará `$` em vez de levantar *`ValueError`*.

Embora outras exceções ainda possam ocorrer, esse método é chamado de "seguro" porque sempre tenta retornar uma string utilizável em vez de levantar uma exceção. Em outro sentido, *`safe_substitute()`* pode ser qualquer coisa diferente de seguro, uma vez que irá ignorar silenciosamente modelos malformados contendo delimitadores pendentes, chaves não correspondidas ou espacos reservados que não são identificadores Python válidos.

Instâncias de *`Template`* também fornecem um atributo de dados públicos:

template
Este é o objeto passado para o argumento *template* do construtor. Em geral, você não deve alterá-lo, mas o acesso somente leitura não é obrigatório.

Aqui está um exemplo de como usar uma instância de Template:

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
```

(continua na próxima página)

(continuação da página anterior)

```
ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

Uso avançado: você pode derivar subclasses de *Template* para personalizar a sintaxe do espaço reservado, caractere delimitador ou toda a expressão regular usada para analisar strings de modelo. Para fazer isso, você pode substituir estes atributos de classe:

- *delimiter* – Este é a string literal que descreve um delimitador de introdução do espaço reservado. O valor padrão é `$`. Note que esta *não* deve ser uma expressão regular, já que a implementação irá chamar *re.escape()* nesta string conforme necessário. Observe também que você não pode alterar o delimitador após a criação da classe (ou seja, um delimitador diferente deve ser definido no espaço de nomes da classe da subclasse).
- *idpattern* – Esta é a expressão regular que descreve o padrão para espaço reservado sem envolto em chaves. O valor padrão é a expressão regular `(?a:[_a-z][_a-z0-9]*)`. Se for fornecido e *braceidpattern* for `None`, esse padrão também se aplicará o espaço reservado com chaves.

  Nota: Uma vez que *flags* padrão é `re.IGNORECASE`, o padrão `[a-z]` pode corresponder a alguns caracteres não ASCII. É por isso que usamos o sinalizador local `a` aqui.

  Alterado na versão 3.7: *braceidpattern* pode ser usado para definir padrões separados usados dentro e fora das chaves.
- *braceidpattern* – É como *idpattern*, mas descreve o padrão para espaços reservados com chaves. O padrão é `None`, o que significa recorrer a *idpattern* (ou seja, o mesmo padrão é usado dentro e fora das chaves). Se fornecido, permite definir padrões diferentes para espaço reservado com e sem chaves.

  Novo na versão 3.7.
- *flags* – Os sinalizadores de expressão regular que serão aplicados ao compilar a expressão regular usada para reconhecer substituições. O valor padrão é `re.IGNORECASE`. Note que `re.VERBOSE` sempre será adicionado aos sinalizadores, então *idpattern*s personalizados devem seguir as convenções para expressões regulares verbosas.

  Novo na versão 3.2.

Como alternativa, você pode fornecer todo o padrão de expressão regular substituindo o atributo *pattern* de classe. Se você fizer isso, o valor deve ser um objeto de expressão regular com quatro grupos de captura nomeados. Os grupos de captura correspondem às regras fornecidas acima, junto com a regra inválida do espaço reservado:

- *escaped* – Este grupo corresponde à sequência de escape, por exemplo `$$`, no padrão.
- *named* – Este grupo corresponde ao nome do espaço reservado sem chaves; não deve incluir o delimitador no grupo de captura.
- *braced* – Este grupo corresponde ao nome do espaço reservado entre chaves; ele não deve incluir o delimitador ou chaves no grupo de captura.
- *invalid* – Esse grupo corresponde a qualquer outro padrão de delimitador (geralmente um único delimitador) e deve aparecer por último na expressão regular.

6.1.5 Funções auxiliares

`string.`**`capwords`**(*s*, *sep=None*)

Divide o argumento em palavras usando `str.split()`, coloca cada palavra em maiúscula usando `str.capitalize()`, e junte as palavras em maiúsculas usando `str.join()`. Se o segundo argumento opcional *sep* estiver ausente ou `None`, os caracteres de espaço em branco são substituídos por um único espaço e os espaços em branco à esquerda e à direita são removidos, caso contrário *sep* é usado para dividir e unir as palavras.

6.2 `re` — Operações com expressões regulares

Código Fonte: Lib/re.py

Este módulo fornece operações de correspondência de expressões regulares semelhantes às encontradas em Perl. O nome do módulo vem da abreviação do termo em inglês *regular expressions*, RE.

Ambos os padrões e strings a serem pesquisados podem ser strings Unicode (`str`) assim como strings de 8 bits (`bytes`). No entanto, strings Unicode e strings de 8 bits não podem ser misturadas: ou seja, você não pode combinar uma string Unicode com um padrão de bytes ou vice-versa; da mesma forma, ao solicitar uma substituição, a string de substituição deve ser do mesmo tipo que o padrão e a string de pesquisa.

Regular expressions use the backslash character (`'\'`) to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python's usage of the same character for the same purpose in string literals; for example, to match a literal backslash, one might have to write `'\\\\'` as the pattern string, because the regular expression must be `\\`, and each backslash must be expressed as `\\` inside a regular Python string literal.

A solução é usar a notação de string bruta do Python para padrões de expressão regular; as contrabarras não são tratadas de nenhuma maneira especial em uma string literal com o prefixo `'r'`. Portanto, `r"\n"` é uma string de dois caracteres contendo `'\'` e `'n'`, enquanto `"\n"` é uma string de um caractere contendo um nova linha. Normalmente, os padrões serão expressos em código Python usando esta notação de string bruta.

É importante notar que a maioria das operações de expressão regular estão disponíveis como funções e métodos de nível de módulo em *expressões regulares compiladas*. As funções são atalhos que não exigem que você compile um objeto regex primeiro, mas perdem alguns parâmetros de ajuste fino.

Ver também:

O módulo regex de terceiros, que possui uma API compatível com o módulo da biblioteca padrão `re`, mas oferece funcionalidades adicionais e um suporte mais completo a Unicode.

6.2.1 Sintaxe de expressão regular

Uma expressão regular (ou ER) especifica um conjunto de strings que corresponde a ela; as funções neste módulo permitem que você verifique se uma determinada string corresponde a uma determinada expressão regular (ou se uma determinada expressão regular corresponde a uma determinada string, o que resulta na mesma coisa).

As expressões regulares podem ser concatenadas para formar novas expressões regulares; se *A* e *B* forem expressões regulares, então *AB* também será uma expressão regular. Em geral, se uma string *p* corresponder a *A* e outra string *q* corresponder a *B*, a string *pq* corresponderá a AB. Isso é válido, a menos que *A* ou *B* contenham operações de baixa precedência; condições de contorno entre *A* e *B*; ou ter referências de grupo numeradas. Assim, expressões complexas podem ser facilmente construídas a partir de expressões primitivas mais simples, como as descritas aqui. Para obter detalhes sobre a teoria e implementação de expressões regulares, consulte o livro de Friedl [Frie09], ou quase qualquer livro sobre construção de compiladores.

Segue uma breve explicação do formato das expressões regulares. Para mais informações e uma apresentação mais suave, consulte o regex-howto.

As expressões regulares podem conter caracteres especiais e comuns. A maioria dos caracteres comuns, como `'A'`, `'a'` ou `'0'`, são as expressões regulares mais simples; eles simplesmente se combinam. Você pode concatenar caracteres comuns, de forma que `último` corresponda à string `'último'`. (No restante desta seção, escreveremos ERs `neste estilo especial`, geralmente sem aspas, e strings para serem correspondidas `'entre aspas simples'`.)

Alguns caracteres, como `'|'` ou `'('`, são especiais. Os caracteres especiais representam classes de caracteres comuns ou afetam como as expressões regulares em torno deles são interpretadas.

Qualificadores de repetição (`*`, `+`, `?`, `{m,n}` etc) não podem ser aninhados diretamente. Isso evita ambiguidade com o sufixo modificador não guloso `?`, E com outros modificadores em outras implementações. Para aplicar uma segunda repetição a uma repetição interna, podem ser usados parênteses. Por exemplo, a expressão `(?:a{6})*` corresponde a qualquer múltiplo de seis caracteres `'a'`.

Os caracteres especiais são:

. (Ponto.) No modo padrão, corresponde a qualquer caractere, exceto uma nova linha. Se o sinalizador *DOTALL* foi especificado, ele corresponde a qualquer caractere, incluindo uma nova linha.

^ (Sinal de circunflexo.) Corresponde ao início da string, e no modo *MULTILINE* também corresponde imediatamente após cada nova linha.

$ Corresponde ao final da string ou logo antes da nova linha no final da string, e no modo *MULTILINE* também corresponde antes de uma nova linha. `foo` corresponde a 'foo' e 'foobar', enquanto a expressão regular `foo$` corresponde apenas a 'foo'. Mais interessante, pesquisar por `foo.$` em `'foo1\nfoo2\n'` corresponde a 'foo2' normalmente, mas 'foo1' no modo *MULTILINE*; procurando por um único `$` em `'foo\n'` encontrará duas correspondências (vazias): uma logo antes da nova linha e uma no final da string.

***** Faz com que a ER resultante corresponda a 0 ou mais repetições da ER anterior, tantas repetições quantas forem possíveis. `ab*` corresponderá a 'a', 'ab' ou 'a' seguido por qualquer número de 'b's.

+ Faz com que a ER resultante corresponda a 1 ou mais repetições da ER anterior. `ab+` irá corresponder a 'a' seguido por qualquer número diferente de zero de 'b's; não corresponderá apenas a 'a'.

? Faz com que a ER resultante corresponda a 0 ou 1 repetição da ER anterior. `ab?` irá corresponder a 'a' ou 'ab'.

***?, +?, ??** Os qualificadores `'*'`, `'+'` e `'?'` são todos *gananciosos*; eles correspondem ao máximo de texto possível. Às vezes, esse comportamento não é desejado; se a ER `<.*>` for correspondida com `'<a> b <c>'`, ele irá corresponder a toda a string, e não apenas `'<a>'`. Adicionar `?` após o qualificador faz com que ele execute a correspondência da maneira *não gananciosa* ou *minimalista*; tão *poucos* caracteres quanto possível serão correspondidos. Usando a `<.*?>` irá corresponder apenas `'<a>'`.

{m} Especifica que exatamente *m* cópias da ER anterior devem ser correspondidas; menos correspondências fazem com que toda a ER não seja correspondida. Por exemplo, `a{6}` irá corresponder exatamente a seis caracteres `'a'`, mas não a cinco.

{m,n} Faz com que a ER resultante corresponda de *m* a *n* repetições da ER precedente, tentando corresponder ao máximo de repetições possível. Por exemplo, `a{3,5}` irá corresponder de 3 a 5 caracteres `'a'`. A omissão de *m* especifica um limite inferior de zero e a omissão de *n* especifica um limite superior infinito. Como exemplo, `a{4,}b` irá corresponder a `'aaaab'` ou mil caracteres `'a'` seguidos por um `'b'`, mas não `'aaab'`. A vírgula não pode ser omitida ou o modificador será confundido com a forma descrita anteriormente.

{m,n}? Faz com que a ER resultante corresponda de *m* a *n* repetições da ER precedente, tentando corresponder o mínimo de *poucas* repetições possível. Esta é a versão não gananciosa do qualificador anterior. Por exemplo, na string de 6 caracteres `'aaaaaa'`, `a{3,5}` irá corresponder a 5 caracteres `'a'`, enquanto `a{3,5}?` corresponderá apenas a 3 caracteres.

**** Ou escapa caracteres especiais (permitindo que você corresponde a caracteres como `'*'`, `'?'` e assim por diante), ou sinaliza uma sequência especial; sequências especiais são discutidas abaixo.

Se você não estiver usando uma string raw para expressar o padrão, lembre-se de que o Python também usa a contrabarra como uma sequência de escape em literais de string; se a sequência de escape não for reconhecida pelo analisador sintático do Python, a contrabarra e o caractere subsequente serão incluídos na string resultante. No entanto, se o Python reconhecer a sequência resultante, a contrabarra deve ser repetida duas vezes. Isso é complicado e difícil de entender, portanto, é altamente recomendável que você use strings raw para todas as expressões, exceto as mais simples.

[] Usado para indicar um conjunto de caracteres. Em um conjunto:

- Caracteres podem ser listados individualmente, por exemplo, `[amk]` vai corresponder a `'a'`, `'m'` ou `'k'`.
- Intervalos de caracteres podem ser indicados fornecendo dois caracteres e separando-os por `'-'`, por exemplo `[a-z]` irá corresponder a qualquer letra ASCII minúscula, `[0-5][0-9]` irá corresponder a todos os números de dois dígitos de `00` a `59`, e `[0-9A-Fa-f]` irá corresponder a qualquer dígito hexadecimal. Se `-` for escapado (por exemplo, `[a\-z]`) ou se for colocado como o primeiro ou último caractere (por exemplo, `[-a]` ou `[a-]`), ele corresponderá a um literal `'-'`.
- Os caracteres especiais perdem seu significado especial dentro dos conjuntos. Por exemplo, `[(+*)]` corresponderá a qualquer um dos caracteres literais `'('`, `'+'`, `'*'` ou `')'`.
- Classes de caracteres como `\w` ou `\S` (definidas abaixo) também são aceitas dentro de um conjunto, embora os caracteres que correspondem dependam do modo *ASCII* ou *LOCALE* está em vigor.
- Os caracteres que não estão dentro de um intervalo podem ser correspondidos *complementando* o conjunto. Se o primeiro caractere do conjunto for `'^'`, todos os caracteres que *não* estiverem no conjunto serão correspondidos. Por exemplo, `[^5]` irá corresponder a qualquer caractere exceto `'5'`, e `[^^]` irá corresponder a qualquer caractere exceto `'^'`. `^` não tem nenhum significado especial se não for o primeiro caractere do conjunto.
- Para corresponder a um `']'` literal dentro de um conjunto, preceda-o com uma contrabarra ou coloque-o no início do conjunto. Por exemplo, `[()[\]{}]` e `[]()[{}]` ambos corresponderão a um parêntese.
- Suporte de conjuntos aninhados e operações de conjunto como no Unicode Technical Standard #18 podem ser adicionados no futuro. Isso mudaria a sintaxe, então para facilitar essa mudança uma *FutureWarning* será gerado em casos ambíguos por enquanto. Isso inclui conjuntos que começam com um `'['` literal ou contendo sequências de caracteres literais `'--'`, `'&&'`, `'~~'` e `'||'`. Para evitar um aviso, escape-os com uma contrabarra.

Alterado na versão 3.7: *FutureWarning* é levantado se um conjunto de caracteres contém construções que mudarão semanticamente no futuro.

| `A|B`, onde *A* e *B* podem ser ERs arbitrários, cria uma expressão regular que corresponderá a *A* ou *B*. Um número arbitrário de ERs pode ser separado por `'|'` desta forma. Isso também pode ser usado dentro de grupos (veja abaixo). Conforme a string alvo é escaneada, ERs separados por `'|'` são tentados da esquerda para a direita. Quando um padrão corresponde completamente, essa ramificação é aceita. Isso significa que, assim que *A* corresponder, *B* não será testado posteriormente, mesmo que produza uma correspondência geral mais longa. Em outras palavras, o operador `'|'` nunca é ganancioso. Para corresponder a um `'|'` literal, use `\|`, ou coloque-o dentro de uma classe de caractere, como em `[|]`.

(...) Corresponde a qualquer expressão regular que esteja entre parênteses e indica o início e o fim de um grupo; o conteúdo de um grupo pode ser recuperado após uma correspondência ter sido realizada e pode ser correspondido posteriormente na string com a sequência especial `\number`, descrita abaixo. Para corresponder aos literais `'('` ou `')'`, use `\(` ou `\)`, ou coloque-os dentro de uma classe de caracteres: `[(]`, `[)]`.

(?...) Esta é uma notação de extensão (um `'?'` seguindo um `'('` não é significativo de outra forma). O primeiro caractere após o `'?'` determina qual o significado e sintaxe posterior do é. As extensões normalmente não criam um novo grupo; `(?P<name>...)` é a única exceção a esta regra. A seguir estão as extensões atualmente suportadas.

(?aiLmsux) (Uma ou mais letras do conjunto `'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'`.) O grupo corresponde à string vazia; as letras definem os sinalizadores correspondentes: *re.A* (correspondência somente ASCII), *re.I* (não

diferenciar maiúsculas e minúsculas), *re.L* (dependente do local), *re.M* (multi-linha), *re.S* (ponto corresponde a todos), re.U (correspondência Unicode) e *re.X* (detalhado), para toda a expressão regular. (Os sinalizadores são descritos em *Conteúdo do Módulo*.) Isso é útil se você deseja incluir os sinalizadores como parte da expressão regular, em vez de passar um argumento *flag* para função *re.compile()*. Os sinalizadores devem ser usados primeiro na string de expressão.

(?:...) Uma versão sem captura de parênteses regulares. Corresponde a qualquer expressão regular que esteja entre parênteses, mas a substring correspondida pelo grupo *não pode* ser recuperada após realizar uma correspondência ou referenciada posteriormente no padrão.

(?aiLmsux-imsx:...) (Zero ou mais letras de o conjunto 'a', 'i', 'L', 'm', 's', 'u', 'x', opcionalmente seguido por '-' seguido por uma ou mais letras de 'i', 'm', 's', 'x'.) As letras definem ou removem os sinalizadores correspondentes: *re.A* (correspondência somente ASCII), *re.I* (sem diferenciar maiúsculas e minúsculas), *re.L* (dependente do local), *re.M* (multi-linhas), *re.S* (ponto corresponde a todos), re.U (correspondência Unicode) e *re.X* (detalhamento), para a parte da expressão. (Os sinalizadores são descritos em *Conteúdo do Módulo*.)

As letras 'a', 'L' e 'u' são mutuamente exclusivas quando usadas como sinalizadores em linha, portanto, não podem ser correspondidas ou seguir '-'. Em vez disso, quando um deles aparece em um grupo embutido, ele substitui o modo de correspondência no grupo anexo. Em padrões Unicode (?a:...) muda para correspondência somente ASCII, e (?u:...) muda para correspondência Unicode (padrão). No padrão de byte (?L:...) muda para a localidade dependendo da correspondência, e (?a:...) muda para correspondência apenas ASCII (padrão). Esta substituição só tem efeito para o grupo estreito em linha e o modo de correspondência original é restaurado fora do grupo.

Novo na versão 3.6.

Alterado na versão 3.7: As letras 'a', 'L' e 'u' também podem ser usadas em um grupo.

(?P<name>...) Semelhante aos parênteses regulares, mas a substring correspondida pelo grupo é acessível por meio do nome de grupo simbólico *name*. Os nomes de grupo devem ser identificadores Python válidos e cada nome de grupo deve ser definido apenas uma vez em uma expressão regular. Um grupo simbólico também é um grupo numerado, como se o grupo não tivesse um nome.

Grupos nomeados podem ser referenciados em três contextos. Se o padrão for (?P<quote>['"]).*?(?P=quote) (ou seja, corresponder a uma string entre aspas simples ou duplas):

Contexto de referência ao grupo "quote"	Formas de referenciá-lo
no mesmo padrão	• (?P=quote) (como mostrado) • \1
ao processar a correspondência do objeto *m*	• m.group('quote') • m.end('quote') (etc.)
em uma string passada para o argumento *repl* de re.sub()	• \g<quote> • \g<1> • \1

(?P=name) Uma referência anterior a um grupo nomeado; corresponde a qualquer texto que corresponda ao grupo anterior denominado *name*.

(?#...) Um comentário; o conteúdo dos parênteses é simplesmente ignorado.

(?=...) Corresponde se ... corresponder a próxima, mas não consome nada da string. Isso é chamado de *asserção lookahead*. Por exemplo, Isaac (?=Asimov) corresponderá a 'Isaac ' apenas se for seguido por

'Asimov'.

(?!...) Corresponde se `...` não corresponder a próxima. Isso é uma *asserção lookahead negativa*. Por exemplo, `Isaac (?!Asimov)` corresponderá a `'Isaac '` apenas se *não* for seguido por `'Asimov'`.

(?<=...) Corresponde se a posição atual na string for precedida por uma correspondência para `...` que termina na posição atual. Isso é chamado de *asserção retrovisora positiva*. `(?<=abc)def` irá encontrar uma correspondência em `'abcdef'`, uma vez que o retrovisor vai voltar 3 caracteres e verificar se o padrão contido corresponde. O padrão contido deve corresponder apenas a strings de algum comprimento fixo, o que significa que `abc` ou `a|b` são permitidos, mas `a*` e `a{3,4}` não são. Observe que os padrões que começam com asserções retrovisoras positivas não corresponderão ao início da string que está sendo pesquisada; você provavelmente desejará usar a função *search()* em vez da função *match()*:

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

Este exemplo procura por uma palavra logo após um hífen:

```
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

Alterado na versão 3.5: Adicionado suporte para referências de grupo de comprimento fixo.

(?<!...) Corresponde se a posição atual na string não for precedida por uma correspondência para `...`. Isso é chamado de *asserção retrovisora negativa*. Semelhante às asserções retrovisoras positivas, o padrão contido deve corresponder apenas a strings de algum comprimento fixo. Os padrões que começam com asserções retroativas negativas podem corresponder ao início da string que está sendo pesquisada.

(?(id/name)yes-pattern|no-pattern) Tentará corresponder com `padrão-sim` se o grupo com determinado *id* ou *nome* existir, e com `padrão-não` se não existir. `padrão-não` é opcional e pode ser omitido. Por exemplo, `(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)` é um padrão ruim de correspondência de e-mail, que corresponderá com `'<usuario@host.com>'` bem como `'usuario@host.com'`, mas não com `'<usuario@host.com>'` nem `'usuario@host.com>'`.

As sequências especiais consistem em `'\'` e um caractere da lista abaixo. Se o caractere comum não for um dígito ASCII ou uma letra ASCII, a ER resultante corresponderá ao segundo caractere. Por exemplo, `\$` corresponde ao caractere `'$'`.

\number Corresponde ao conteúdo do grupo de mesmo número. Os grupos são numerados a partir de 1. Por exemplo, `(.+) \1` corresponde a `'de de'` ou `'55 55'`, mas não `'dede'` (note o espaço após o grupo). Esta sequência especial só pode ser usada para corresponder a um dos primeiros 99 grupos. Se o primeiro dígito de *número* for 0, ou *número* tiver 3 dígitos octais de comprimento, ele não será interpretado como uma correspondência de grupo, mas como o caractere com *número* de valor octal. Dentro de `'['` e `']'` de uma classe de caracteres, todos os escapes numéricos são tratados como caracteres.

\A Corresponde apenas ao início da string.

\b Corresponde à string vazia, mas apenas no início ou no final de uma palavra. Uma palavra é definida como uma sequência de caracteres de palavras. Observe que, formalmente, `\b` é definido como a fronteira entre um caractere `\w` e um `\W` (ou vice-versa), ou entre `\w` e o início/fim da string. Isso significa que `r'\bfoo\b'` corresponde a `'foo'`, `'foo.'`, `'(foo)'`, `'bar foo baz'`, mas não a `'foobar'` ou `'foo3'`.

Por padrão, os alfanuméricos Unicode são aqueles usados nos padrões Unicode, mas isso pode ser alterado usando o sinalizador *ASCII*. Os limites das palavras são determinados pela localidade atual se o sinalizador *LOCALE* for usado. Dentro de um intervalo de caracteres, `\b` representa o caractere de backspace, para compatibilidade com os literais de string do Python.

\B Corresponde à string vazia, mas apenas quando *não* estiver no início ou no final de uma palavra. Isso significa que `r'py\B'` corresponde a `'python'`, `'py3'`, `'py2'`, mas não `'py'`, `'py.'`, or `'py!'`. `\B` é exatamente o oposto de `\b`, então caracteres de palavras em padrões Unicode são alfanuméricos Unicode ou o sublinhado, embora isso possa ser alterado usando o sinalizador *ASCII*. Os limites das palavras são determinados pela localidade atual se o sinalizador *LOCALE* for usado.

\d

Para padrões (str) Unicode: Corresponde a qualquer dígito decimal Unicode (ou seja, qualquer caractere na categoria de caractere Unicode [Nd]). Isso inclui `[0-9]`, e também muitos outros caracteres de dígitos. Se o sinalizador *ASCII* for usado, apenas `[0-9]` será correspondido.

Para padrões de 8 bits (isto é, bytes): Corresponde a qualquer dígito decimal; isso é equivalente a `[0-9]`.

\D Corresponde a qualquer caractere que não seja um dígito decimal. Este é o oposto de `\d`. Se o sinalizador *ASCII* for usado, isso se tornará o equivalente a `[^0-9]`.

\s

Para padrões (str) Unicode: Corresponde a caracteres de espaço em branco Unicode (que inclui `[ \t\n\r\f\v]`, e também muitos outros caracteres, como, por exemplo, os espaços não separáveis exigidos pelas regras de tipografia em muitos idiomas). Se o sinalizador *ASCII* for usado, apenas `[ \t\n\r\f\v]` é correspondido.

Para padrões de 8 bits (isto é, bytes): Corresponde a caracteres considerados espaços em branco no conjunto de caracteres ASCII; isso é equivalente a `[ \t\n\r\f\v]`.

\S Corresponde a qualquer caractere que não seja um caractere de espaço em branco. Este é o oposto de `\s`. Se o sinalizador *ASCII* for usado, isso se tornará o equivalente a `[^ \t\n\r\f\v]`.

\w

Para padrões (str) Unicode: Corresponde a caracteres de palavras Unicode; isso inclui a maioria dos caracteres que podem fazer parte de uma palavra em qualquer idioma, bem como números e sublinhado. Se o sinalizador *ASCII* for usado, apenas `[a-zA-Z0-9_]` será correspondido.

Para padrões de 8 bits (isto é, bytes): Corresponde a caracteres considerados alfanuméricos no conjunto de caracteres ASCII; isso é equivalente a `[a-zA-Z0-9_]`. Se o sinalizador *LOCALE* for usado, corresponde aos caracteres considerados alfanuméricos na localidade atual e o sublinhado.

\W Corresponde a qualquer caractere que não seja um caractere de palavra. Este é o oposto de `\w`. Se o sinalizador *ASCII* for usado, ele se tornará o equivalente a `[^a-zA-Z0-9_]`. Se o sinalizador *LOCALE* for usado, corresponde a caracteres que não são alfanuméricos na localidade local atual nem o sublinhado.

\Z Corresponde apenas ao final da string.

A maioria dos escapes padrão suportados por literais de string Python também são aceitos pelo analisador sintático de expressão regular:

```
\a      \b      \f      \n
\r      \t      \u      \U
\v      \x      \\
```

(Observe que `\b` é usado para representar limites de palavras e significa fazer "backspace" apenas dentro das classes de caracteres.)

`'\u'` and `'\U'` escape sequences are only recognized in Unicode patterns. In bytes patterns they are errors. Unknown escapes of ASCII letters are reserved for future use and treated as errors.

Os escapes octais são incluídos em um formulário limitado. Se o primeiro dígito for 0, ou se houver três dígitos octais, é considerado um escape octal. Caso contrário, é uma referência de grupo. Quanto aos literais de string, os escapes octais têm sempre no máximo três dígitos.

Alterado na versão 3.3: As sequências de escape `'\u'` e `'\U'` foram adicionadas.

Alterado na versão 3.6: Escapes desconhecidos consistindo em `'\'` e uma letra ASCII agora são erros.

6.2.2 Conteúdo do Módulo

O módulo define várias funções, constantes e uma exceção. Algumas das funções são versões simplificadas dos métodos completos para expressões regulares compiladas. A maioria dos aplicativos não triviais sempre usa a forma compilada.

Alterado na versão 3.6: Constantes de sinalizadores são agora instâncias de `RegexFlag`, que é uma subclasse de *enum.IntFlag*.

re.**compile**(*pattern*, *flags=0*)

Compila um padrão de expressão regular em um *objeto de expressão regular*, que pode ser usado para correspondência usando seu *match()*, *search()* e outros métodos, descritos abaixo.

O comportamento da expressão pode ser modificado especificando um valor *flags*. Os valores podem ser qualquer uma das seguintes variáveis, correspondidas usando OU bit a bit (o operador `|`).

A sequência

```
prog = re.compile(pattern)
result = prog.match(string)
```

é equivalente a:

```
result = re.match(pattern, string)
```

mas usar *re.compile()* e salvar o objeto de expressão regular resultante para reutilização é mais eficiente quando a expressão será usada várias vezes em um único programa.

Nota: As versões compiladas dos padrões mais recentes passados para *re.compile()* e as funções de correspondência em nível de módulo são armazenadas em cache, de modo que programas que usam apenas algumas expressões regulares por vez não precisam se preocupar em compilar expressões regulares.

---

re.**A**
re.**ASCII**

Faz com que `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` e `\S` executem a correspondência somente ASCII em vez da correspondência Unicode completa. Isso é significativo apenas para padrões Unicode e é ignorado para padrões de bytes. Corresponde ao sinalizador em linha `(?a)`.

Observe que, para compatibilidade com versões anteriores, o sinalizador `re.U` ainda existe (bem como seu sinônimo `re.UNICODE` e sua contraparte incorporada `(?u)`), mas estes são redundantes em Python 3, pois as correspondências são Unicode por padrão para strings (e a correspondência Unicode não é permitida para bytes).

re.**DEBUG**

Exibe informações de depuração sobre a expressão compilada. Nenhum sinalizador em linha correspondente.

re.**I**
re.**IGNORECASE**

Executa uma correspondência que não diferencia maiúsculas de minúsculas; expressões como `[A-Z]` também corresponderão a letras minúsculas. A correspondência Unicode completa (como Ü correspondendo a ü) também funciona, a menos que o sinalizador *re.ASCII* seja usado para desabilitar correspondências não ASCII. A localidade atual não muda o efeito deste sinalizador a menos que o sinalizador *re.LOCALE* também seja usado. Corresponde ao sinalizador em linha `(?i)`.

Observe que quando os padrões Unicode `[a-z]` ou `[A-Z]` são usados em combinação com o sinalizador *IGNORECASE*, eles corresponderão às 52 letras ASCII e 4 letras não ASCII adicionais: 'İ' (U+0130, letra latina I maiúscula com ponto em cima), 'ı' (U+0131, letra latina i minúscula sem ponto), 'ſ' (U+017F, letra latina s minúscula longa) e 'K' (U+212A, sinal de Kelvin). Se o sinalizador *ASCII* for usado, apenas as letras 'a' a 'z' e 'A' a 'Z' serão correspondidas.

re.**L**
re.**LOCALE**
Faz `\w`, `\W`, `\b`, `\B` e a correspondência sem diferença entre maiúsculas e minúsculas dependente do local atual. Este sinalizador pode ser usado apenas com padrões de bytes. O uso desse sinalizador é desencorajado porque o mecanismo de localidade não é confiável, ele só lida com uma "cultura" por vez e só funciona com localidades de 8 bits. A correspondência Unicode já está habilitada por padrão no Python 3 para padrões Unicode (str) e é capaz de lidar com diferentes localidades/idiomas. Corresponde ao sinalizador em linha `(?L)`.

Alterado na versão 3.6: *re.LOCALE* pode ser usado apenas com padrões de bytes e não é compatível com *re.ASCII*.

Alterado na versão 3.7: Objetos de expressão regular compilados com o sinalizador *re.LOCALE* não dependem mais da localidade em tempo de compilação. Apenas a localidade no momento da correspondência afeta o resultado da correspondência.

re.**M**
re.**MULTILINE**
Quando especificado, o caractere padrão `'^'` corresponde ao início da string e ao início de cada linha (imediatamente após cada nova linha); e o caractere padrão `'$'` corresponde ao final da string e ao final de cada linha (imediatamente antes de cada nova linha). Por padrão, `'^'` corresponde apenas no início da string, e `'$'` apenas no final da string e imediatamente antes da nova linha (se houver) no final da string. Corresponde ao sinalizador em linha `(?m)`.

re.**S**
re.**DOTALL**
Faz o caractere especial `'.'` corresponder a qualquer caractere, incluindo uma nova linha; sem este sinalizador, `'.'` irá corresponder a qualquer coisa *exceto* uma nova linha. Corresponde ao sinalizador em linha `(?s)`.

re.**X**
re.**VERBOSE**
Este sinalizador permite que você escreva expressões regulares que parecem mais agradáveis e são mais legíveis, permitindo que você separe visualmente seções lógicas do padrão e adicione comentários. O espaço em branco dentro do padrão é ignorado, exceto quando em uma classe de caractere, ou quando precedido por uma contrabarra sem escape, ou dentro de tokens como `*?`, `(?:` ou `(?P<...>`. Quando uma linha contém um # que não está em uma classe de caractere e não é precedido por uma contrabarra sem escape, todos os caracteres da extremidade esquerda, como # até o final da linha são ignorados.

Isso significa que os dois seguintes objetos de expressão regular que correspondem a um número decimal são funcionalmente iguais:

```
a = re.compile(r"""\d +  # the integral part
                   \.    # the decimal point
                   \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

Corresponde ao sinalizador em linha `(?x)`.

re.**search**(*pattern*, *string*, *flags=0*)
Percorre a *string* procurando o primeiro local onde o padrão *pattern* de expressão regular produz uma correspondência e retorna um *objeto de correspondência* encontrado. Retorne `None` se nenhuma posição na string corresponder ao padrão; observe que isso é diferente de encontrar uma correspondência de comprimento zero em algum ponto da string.

re.**match**(*pattern*, *string*, *flags=0*)

Se zero ou mais caracteres no início da *string* corresponderem ao padrão *pattern* da expressão regular, retorna um *objeto de correspondência* encontrado. Retorna `None` se a string não corresponder ao padrão; observe que isso é diferente de uma correspondência de comprimento zero.

Observe que mesmo no modo `MULTILINE`, `re.match()` irá corresponder apenas no início da string e não no início de cada linha.

Se você quiser localizar uma correspondência em qualquer lugar em *string*, use `search()` (veja também *search() vs. match()*).

re.**fullmatch**(*pattern*, *string*, *flags=0*)

Se toda a *string* corresponder ao padrão *pattern* da expressão regular, retorna um *objeto de correspondência* encontrado. Retorna `None` se a string não corresponder ao padrão; observe que isso é diferente de uma correspondência de comprimento zero.

Novo na versão 3.4.

re.**split**(*pattern*, *string*, *maxsplit=0*, *flags=0*)

Divide a *string* pelas ocorrências do padrão *pattern*. Se parênteses de captura forem usados em *pattern*, o texto de todos os grupos no padrão também será retornado como parte da lista resultante. Se *maxsplit* for diferente de zero, no máximo *maxsplit* divisões ocorrerão e o restante da string será retornado como o elemento final da lista.

```
>>> re.split(r'\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split(r'\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

Se houver grupos de captura no separador e ele corresponder ao início da string, o resultado começará com uma string vazia. O mesmo vale para o final da string:

```
>>> re.split(r'(\W+)', '...words, words...')
['', '...', 'words', ', ', 'words', '...', '']
```

Dessa forma, os componentes do separador são sempre encontrados nos mesmos índices relativos na lista de resultados.

As correspondências vazias para o padrão dividem a string apenas quando não adjacente a uma correspondência vazia anterior.

```
>>> re.split(r'\b', 'Words, words, words.')
['', 'Words', ', ', 'words', ', ', 'words', '.']
>>> re.split(r'\W*', '...words...')
['', '', 'w', 'o', 'r', 'd', 's', '', '']
>>> re.split(r'(\W*)', '...words...')
['', '...', '', '', 'w', '', 'o', '', 'r', '', 'd', '', 's', '...', '', '', '']
```

Alterado na versão 3.1: Adicionado o argumento de sinalizadores opcionais.

Alterado na versão 3.7: Adicionado suporte de divisão em um padrão que pode corresponder a uma string vazia.

re.**findall**(*pattern*, *string*, *flags=0*)

Retorna todas as correspondências não sobrepostas de padrão *pattern* na *string*, como uma lista de strings. A *string* é verificada da esquerda para a direita e as correspondências são retornadas na ordem encontrada. Se um ou mais grupos estiverem presentes no padrão, retorna uma lista de grupos; esta será uma lista de tuplas se o padrão tiver mais de um grupo. Correspondências vazias são incluídas no resultado.

Alterado na versão 3.7: Correspondências não vazias agora podem começar logo após uma correspondência vazia anterior.

re.**finditer**(*pattern*, *string*, *flags=0*)

Retorna um *iterador* produzindo *objetos de correspondência* sobre todas as correspondências não sobrepostas para o padrão *pattern* de ER na *string*. A *string* é percorrida da esquerda para a direita e as correspondências são retornadas na ordem encontrada. Correspondências vazias são incluídas no resultado.

Alterado na versão 3.7: Correspondências não vazias agora podem começar logo após uma correspondência vazia anterior.

re.**sub**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Retorna a string obtida substituindo as ocorrências não sobrepostas da extremidade esquerda do padrão *pattern* na *string* pela substituição *repl*. Se o padrão não for encontrado, *string* será retornado inalterado. *repl* pode ser uma string ou uma função; se for uma string, qualquer escape de contrabarra será processado. Ou seja, `\n` é convertido em um único caractere de nova linha, `\r` é convertido em um retorno de carro e assim por diante. Escapes desconhecidos de letras ASCII são reservados para uso futuro e tratados como erros. Outros escapes desconhecidos como `\&` são deixados como estão. Retrovisores, como `\6`, são substituídos pela substring correspondida pelo grupo 6 no padrão. Por exemplo:

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...         r'static PyObject*\npy_\1(void)\n{',
...         'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

Se *repl* for uma função, ela será chamada para cada ocorrência não sobreposta do padrão *pattern*. A função recebe um único argumento *objeto de correspondência* e retorna a string de substituição. Por exemplo:

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

O padrão pode ser uma string ou um *objeto de padrão*.

O argumento opcional *count* é o número máximo de ocorrências de padrão a serem substituídas; *count* deve ser um número inteiro não negativo. Se omitido ou zero, todas as ocorrências serão substituídas. As correspondências vazias para o padrão são substituídas apenas quando não adjacentes a uma correspondência vazia anterior, então `sub('x*', '-', 'abxd')` retorna `'-a-b--d-'`.

Em argumentos *repl* do tipo string, além dos escapes de caractere e retrovisores descritos acima, `\g<nome>` usará a substring correspondida pelo grupo denominado `nome`, conforme definido pela sintaxe `(?P<nome>...)`. `\g<número>` usa o número do grupo correspondente; `\g<2>` é portanto equivalente a `\2`, mas não é ambíguo em uma substituição como `\g<2>0`. `\20` seria interpretado como uma referência ao grupo 20, não uma referência ao grupo 2 seguida pelo caractere literal `'0'`. O retrovisor `\g6` substitui em toda a substring correspondida pela ER.

Alterado na versão 3.1: Adicionado o argumento de sinalizadores opcionais.

Alterado na versão 3.5: Grupos sem correspondência são substituídos por uma string vazia.

Alterado na versão 3.6: Escapes desconhecidos no padrão *pattern* consistindo em `'\'` e uma letra ASCII agora são erros.

Alterado na versão 3.7: Escapes desconhecidos em *repl* consistindo em `'\'` e uma letra ASCII agora são erros.

Alterado na versão 3.7: As correspondências vazias para o padrão são substituídas quando adjacentes a uma correspondência não vazia anterior.

re.**subn**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)
Executa a mesma operação como *sub()*, mas retorna uma tupla (new_string, number_of_subs_made).

Alterado na versão 3.1: Adicionado o argumento de sinalizadores opcionais.

Alterado na versão 3.5: Grupos sem correspondência são substituídos por uma string vazia.

re.**escape**(*pattern*)
Escape caracteres especiais no padrão *pattern*. Isso é útil se você deseja corresponder uma string literal arbitrária que pode conter metacaracteres de expressão reguladora. Por exemplo:

```
>>> print(re.escape('http://www.python.org'))
http://www\.python\.org

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"
>>> print('[%s]+' % re.escape(legal_chars))
[abcdefghijklmnopqrstuvwxyz0123456789!\#\$%\&\'\*\+\-\.\^_`\|\~:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print('|'.join(map(re.escape, sorted(operators, reverse=True))))
/|\-|\+|\*\*|\*
```

Esta função não deve ser usada para a string de substituição em *sub()* e *subn()*, apenas contrabarras devem ser escapadas. Por exemplo:

```
>>> digits_re = r'\d+'
>>> sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'
>>> print(re.sub(digits_re, digits_re.replace('\\', r'\\'), sample))
/usr/sbin/sendmail - \d+ errors, \d+ warnings
```

Alterado na versão 3.3: O caractere `'_'` não é mais escapada.

Alterado na versão 3.7: Somente caracteres que podem ter um significado especial em uma expressão regular são escapados. Como um resultado,"!', ``'"', '%', "'", ',', '/', ':', ';', '<', '=', '>', '@' e "`" não são mais escapados.

re.**purge**()
Limpa o cache de expressão regular.

exception re.**error**(*msg*, *pattern=None*, *pos=None*)
Exceção levantada quando uma string passada para uma das funções aqui não é uma expressão regular válida (por exemplo, ela pode conter parênteses não correspondentes) ou quando algum outro erro ocorre durante a compilação ou correspondência. Nunca é um erro se uma string não contém correspondência para um padrão. A instância de erro possui os seguintes atributos adicionais:

msg
A mensagem de erro não formatada.

pattern
O padrão da expressão regular.

pos
O índice no padrão *pattern* no qual a compilação falhou (pode ser None).

lineno
A linha correspondente a *pos* (pode ser None).

colno
A coluna correspondente a *pos* (pode ser None).

Alterado na versão 3.5: Adicionados os atributos adicionais.

6.2.3 Objetos expressão regular

Objetos expressão regular compilados oferecem suporte aos seguintes métodos e atributos:

`Pattern.`**`search`**(*string*[, *pos*[, *endpos*]])

Percorre a *string* procurando o primeiro local onde esta expressão regular produz uma correspondência e retorna um *objeto correspondência* encontrado. Retorna `None` se nenhuma posição na string corresponder ao padrão; observe que isso é diferente de encontrar uma correspondência de comprimento zero em algum ponto da string.

O segundo parâmetro opcional *pos* fornece um índice na string onde a pesquisa deve começar; o padrão é `0`. Isso não é totalmente equivalente a fatiar a string; o caractere padrão `'^'` corresponde no início real da string e nas posições logo após uma nova linha, mas não necessariamente no índice onde a pesquisa deve começar.

O parâmetro opcional *endpos* limita o quão longe a string será pesquisada; será como se a string tivesse *endpos* caracteres, então apenas os caracteres de *pos* a `endpos - 1` serão procurados por uma correspondência. Se *endpos* for menor que *pos*, nenhuma correspondência será encontrada; caso contrário, se *rx* é um objeto de expressão regular compilado, `rx.search(string, 0, 50)` é equivalente a `rx.search(string[:50], 0)`.

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # Match at index 0
<re.Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1)  # No match; search doesn't include the "d"
```

`Pattern.`**`match`**(*string*[, *pos*[, *endpos*]])

Se zero ou mais caracteres no *start* da *string* corresponderem a esta expressão regular, retorna um *objeto correspondência* encontrado. Retorna `None` se a string não corresponder ao padrão; observe que isso é diferente de uma correspondência de comprimento zero.

Os parâmetros opcionais *pos* e *endpos* têm o mesmo significado que para o método *`search()`*.

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.match("dog", 1)   # Match as "o" is the 2nd character of "dog".
<re.Match object; span=(1, 2), match='o'>
```

Se você quiser localizar uma correspondência em qualquer lugar em *string*, use *`search()`* ao invés (veja também *search() vs. match()*).

`Pattern.`**`fullmatch`**(*string*[, *pos*[, *endpos*]])

Se toda a *string* corresponder a esta expressão regular, retorna um *objeto correspondência* encontrado. Retorna `None` se a string não corresponder ao padrão; observe que isso é diferente de uma correspondência de comprimento zero.

Os parâmetros opcionais *pos* e *endpos* têm o mesmo significado que para o método *`search()`*.

```
>>> pattern = re.compile("o[gh]")
>>> pattern.fullmatch("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.fullmatch("ogre")     # No match as not the full string matches.
>>> pattern.fullmatch("doggie", 1, 3)   # Matches within given limits.
<re.Match object; span=(1, 3), match='og'>
```

Novo na versão 3.4.

`Pattern.`**`split`**(*string*, *maxsplit=0*)

Idêntico à função *`split()`*, usando o padrão compilado.

`Pattern.`**`findall`**(*string*[, *pos*[, *endpos*]])

Semelhante à função *`findall()`*, usando o padrão compilado, mas também aceita os parâmetros *pos* e *endpos* opcionais que limitam a região de pesquisa como para *`search()`*.

`Pattern.`**`finditer`**(*string*[, *pos*[, *endpos*]])
Semelhante à função `finditer()`, usando o padrão compilado, mas também aceita os parâmetros *pos* e *endpos* opcionais que limitam a região de pesquisa como para `search()`.

`Pattern.`**`sub`**(*repl*, *string*, *count=0*)
Idêntico à função `sub()`, usando o padrão compilado.

`Pattern.`**`subn`**(*repl*, *string*, *count=0*)
Idêntico à função `subn()`, usando o padrão compilado.

`Pattern.`**`flags`**
Os sinalizadores de correspondência de regex. Esta é uma combinação dos sinalizadores fornecidos para `compile()`, qualquer sinalizador em linha `(?...)` no padrão e sinalizadores implícitos como `UNICODE` se o padrão for uma string Unicode.

`Pattern.`**`groups`**
O número de grupos de captura no padrão.

`Pattern.`**`groupindex`**
Um dicionário que mapeia qualquer nome de grupo simbólico definido por `(?P<id>)` para números de grupo. O dicionário estará vazio se nenhum grupo simbólico for usado no padrão.

`Pattern.`**`pattern`**
A string de padrão da qual o objeto de padrão foi compilado.

Alterado na versão 3.7: Adicionado suporte de `copy.copy()` e `copy.deepcopy()`. Os objetos expressão regular compilados são considerados atômicos.

6.2.4 Objetos correspondência

Objetos correspondência sempre têm um valor booleano de `True`. Como `match()` e `search()` retornam `None` quando não há correspondência, você pode testar se houve uma correspondência com uma simples instrução `if`:

```
match = re.search(pattern, string)
if match:
    process(match)
```

Os objetos correspondência oferecem suporte aos seguintes métodos e atributos:

`Match.`**`expand`**(*template*)
Retorna a string obtida fazendo a substituição da contrabarra na string modelo *template*, como feito pelo método `sub()`. Escapes como `\n` são convertidos para os caracteres apropriados, e referências anteriores numéricas (`\1`, `\2`) e retrovisores nomeados (`\g<1>`, `\g<nome>`) são substituídos pelo conteúdo do grupo correspondente.

Alterado na versão 3.5: Grupos sem correspondência são substituídos por uma string vazia.

`Match.`**`group`**([*group1*, ...])
Retorna um ou mais subgrupos da correspondência. Se houver um único argumento, o resultado será uma única string; se houver vários argumentos, o resultado é uma tupla com um item por argumento. Sem argumentos, *group1* padroniza para zero (toda a correspondência é retornada). Se um argumento *groupN* for zero, o valor de retorno correspondente será toda a string correspondente; se estiver no intervalo inclusivo [1..99], é a string que corresponde ao grupo entre parênteses correspondente. Se um número de grupo for negativo ou maior do que o número de grupos definidos no padrão, uma exceção `IndexError` é levantada. Se um grupo estiver contido em uma parte do padrão que não correspondeu, o resultado correspondente será `None`. Se um grupo estiver contido em uma parte do padrão que correspondeu várias vezes, a última correspondência será retornada.

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # The entire match
```

(continua na próxima página)

(continuação da página anterior)

```
'Isaac Newton'
>>> m.group(1)        # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)        # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)     # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

Se a expressão regular usa a sintaxe `(?P<name>...)`, os argumentos *groupN* também podem ser strings que identificam grupos por seus nomes de grupo. Se um argumento string não for usado como um nome de grupo no padrão, uma exceção *IndexError* é levantada.

Um exemplo moderadamente complicado:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

Grupos nomeados também podem ser referidos por seu índice:

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

Se um grupo corresponder várias vezes, apenas a última correspondência estará acessível:

```
>>> m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)                        # Returns only the last match.
'c3'
```

`Match.__getitem__`(*g*)

Isso é idêntico a `m.group(g)`. Isso permite acesso mais fácil a um grupo individual de uma correspondência:

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m[0]       # The entire match
'Isaac Newton'
>>> m[1]       # The first parenthesized subgroup.
'Isaac'
>>> m[2]       # The second parenthesized subgroup.
'Newton'
```

Novo na versão 3.6.

`Match.groups`(*default=None*)

Retorna uma tupla contendo todos os subgrupos da correspondência, de 1 até quantos grupos estiverem no padrão. O argumento *default* é usado para grupos que não participaram da correspondência; o padrão é `None`.

Por exemplo:

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

Se colocarmos a casa decimal e tudo depois dela opcional, nem todos os grupos podem participar da correspondência. Esses grupos serão padronizados como `None`, a menos que o argumento *default* seja fornecido:

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      # Second group defaults to None.
('24', None)
>>> m.groups('0')   # Now, the second group defaults to '0'.
('24', '0')
```

Match.**groupdict**(*default=None*)

Retorna um dicionário contendo todos os subgrupos *named* da correspondência, tendo como chave o nome do subgrupo. O argumento *default* usado para grupos que não participaram da partida; o padrão é `None`. Por exemplo:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

Match.**start**([*group*])

Match.**end**([*group*])

Retorna os índices de início e fim da substring correspondidos pelo grupo *group*; *group* tem como padrão zero (o que significa que toda a substring é correspondida). Retorna `-1` se *group* existe, mas não contribuiu para a correspondência. Para um objeto correspondência *m* e um grupo *g* que contribuiu para a correspondência, a substring correspondida pelo grupo *g* (equivalente a `m.group(g)`) é

```
m.string[m.start(g):m.end(g)]
```

Observe que `m.start(group)` será igual a `m.end(group)` se *group* correspondeu a uma string nula. Por exemplo, após `m = re.search('b(c?)', 'cba')`, `m.start(0)` é 1, `m.end(0)` é 2, `m.start(1)` e `m.end(1)` são 2, e `m.start(2)` levanta uma exceção *IndexError*.

Um exemplo que removerá *remove_this* dos endereços de e-mail:

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

Match.**span**([*group*])

Para uma correspondência *m*, retorna a tupla de dois `(m.start(group), m.end(group))`. Observe que se *group* não contribuiu para a correspondência, isso é `(-1, -1)`. *group* tem como padrão zero, a correspondência inteira.

Match.**pos**

O valor de *pos* que foi passado para o método *search()* ou *match()* de um *objeto de regex*. Este é o índice da string na qual o mecanismo de ER começou a procurar uma correspondência.

Match.**endpos**

O valor de *endpos* que foi passado para o método *search()* ou *match()* de um *objeto de regex*. Este é o índice da string após do qual o mecanismo de ER não vai chegar.

Match.**lastindex**

O índice em número inteiro do último grupo de captura correspondido, ou `None` se nenhum grupo foi correspondido. Por exemplo, as expressões `(a)b`, `((a)(b))` e `((ab))` terão `lastindex == 1` se aplicadas à string `'ab'`, enquanto a expressão `(a)(b)` terá `lastindex == 2`, se aplicada à mesma string.

Match.**lastgroup**

O nome do último grupo de captura correspondido, ou `None` se o grupo não tinha um nome, ou se nenhum grupo foi correspondido.

Match.**re**

O *objeto expressão regular* cujo método *match()* ou *search()* produziu esta instância de correspondência.

Match.**string**
A string passada para *match()* ou *search()*.

Alterado na versão 3.7: Adicionado suporte de *copy.copy()* e *copy.deepcopy()*. Objetos correspondência são considerados atômicos.

6.2.5 Exemplos de expressão regular

Verificando por um par

In this example, we'll use the following helper function to display match objects a little more gracefully:

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

Suponha que você esteja escrevendo um programa de pôquer onde a mão de um jogador é representada como uma string de 5 caracteres com cada caractere representando uma carta, "a" para ás, "k" para rei, "q" para dama, "j" para valete, "t" para 10 e "2" a "9" representando a carta com esse valor.

Para ver se uma determinada string é uma mão válida, pode-se fazer o seguinte:

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q"))  # Valid.
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match("akt5e"))  # Invalid.
>>> displaymatch(valid.match("akt"))    # Invalid.
>>> displaymatch(valid.match("727ak"))  # Valid.
"<Match: '727ak', groups=()>"
```

Essa última mão, `"727ak"`, continha um par, ou duas cartas com o mesmo valor. Para combinar isso com uma expressão regular, pode-se usar retrovisores como:

```
>>> pair = re.compile(r".*(.).*\1")
>>> displaymatch(pair.match("717ak"))     # Pair of 7s.
"<Match: '717', groups=('7',)>"
>>> displaymatch(pair.match("718ak"))     # No pairs.
>>> displaymatch(pair.match("354aa"))     # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```

To find out what card the pair consists of, one could use the *group()* method of the match object in the following manner:

```
>>> pair.match("717ak").group(1)
'7'

# Error because re.match() returns None, which doesn't have a group() method:
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.).*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'

>>> pair.match("354aa").group(1)
'a'
```

Simulando scanf()

Python atualmente não possui um equivalente a `scanf()`. Expressões regulares são geralmente mais poderosas, embora também mais detalhadas, do que strings de formato `scanf()`. A tabela abaixo oferece alguns mapeamentos mais ou menos equivalentes entre os tokens de formato `scanf()` e expressões regulares.

Token `scanf()`	Expressão regular
`%c`	`.`
`%5c`	`.{5}`
`%d`	`[-+]?\d+`
`%e, %E, %f, %g`	`[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?`
`%i`	`[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)`
`%o`	`[-+]?[0-7]+`
`%s`	`\S+`
`%u`	`\d+`
`%x, %X`	`[-+]?(0[xX])?[\dA-Fa-f]+`

Para extrair um nome de arquivo e números de uma string como

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

você usaria um formato de `scanf()` como

```
%s - %d errors, %d warnings
```

A expressão regular equivalente seria

```
(\S+) - (\d+) errors, (\d+) warnings
```

search() vs. match()

Python oferece duas operações primitivas diferentes baseadas em expressões regulares: *re.match()* verifica se há uma correspondência apenas no início da string, enquanto *re.search()* verifica se há uma correspondência em qualquer lugar da string (isto é o que o Perl faz por padrão).

Por exemplo:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("c", "abcdef")   # Match
<re.Match object; span=(2, 3), match='c'>
```

Expressões regulares começando com `'^'` podem ser usadas com *search()* para restringir a correspondência no início da string:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("^c", "abcdef")  # No match
>>> re.search("^a", "abcdef")  # Match
<re.Match object; span=(0, 1), match='a'>
```

Observe, entretanto, que no modo *MULTILINE* *match()* apenas corresponde ao início da string, enquanto que usar *search()* com uma expressão regular começando com `'^'` irá corresponder em no início de cada linha.

```
>>> re.match('X', 'A\nB\nX', re.MULTILINE)  # No match
>>> re.search('^X', 'A\nB\nX', re.MULTILINE)  # Match
<re.Match object; span=(4, 5), match='X'>
```

Criando uma lista telefônica

split() divide uma string em uma lista delimitada pelo padrão passado. O método é inestimável para converter dados textuais em estruturas de dados que podem ser facilmente lidas e modificadas pelo Python, conforme demonstrado no exemplo a seguir que cria uma lista telefônica.

First, here is the input. Normally it may come from a file, here we are using triple-quoted string syntax:

```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```

As entradas são separadas por uma ou mais novas linhas. Agora, convertemos a string em uma lista com cada linha não vazia tendo sua própria entrada:

```
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
'Frank Burger: 925.541.7625 662 South Dogwood Way',
'Heather Albrecht: 548.326.4584 919 Park Place']
```

Finalmente, divida cada entrada em uma lista com nome, sobrenome, número de telefone e endereço. Usamos o parâmetro `maxsplit` de *split()* porque o endereço contém espaços, nosso padrão de divisão:

```
>>> [re.split(":? ", entry, 3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

O padrão `:?` corresponde ao caractere de dois pontos após o sobrenome, de modo que não ocorre na lista de resultados. Com um `maxsplit` de `4`, podemos separar o número da casa do nome da rua:

```
>>> [re.split(":? ", entry, 4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
```

Manipulação de texto

sub() substitui cada ocorrência de um padrão por uma string ou o resultado de uma função. Este exemplo demonstra o uso de *sub()* com uma função para manipular o texto ou aleatorizar a ordem de todos os caracteres em cada palavra de uma frase, exceto o primeiro e o último caracteres:

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'
```

Encontrando todos os advérbios

findall() corresponde a *todas* as ocorrências de um padrão, não apenas a primeira como *search()* faz. Por exemplo, se um escritor deseja encontrar todos os advérbios em algum texto, ele pode usar *findall()* da seguinte maneira:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly", text)
['carefully', 'quickly']
```

Encontrando todos os advérbios e suas posições

Se se deseja obter mais informações sobre todas as correspondências de um padrão do que o texto correspondido, *finditer()* é útil, pois fornece *objetos correspondência* em vez de strings. Continuando com o exemplo anterior, se um escritor quisesse encontrar todos os advérbios *e suas posições* em algum texto, ele usaria *finditer()* da seguinte maneira:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly
```

Notação de string bruta

A notação de string bruta (`r"texto"`) mantém as expressões regulares sãs. Sem ele, cada contrabarra (`'\'`) em uma expressão regular teria que ser prefixada com outra para escapar dela. Por exemplo, as duas linhas de código a seguir são funcionalmente idênticas:

```
>>> re.match(r"\W(.)\1\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
>>> re.match("\\W(.)\\1\\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
```

Quando se deseja corresponder a uma contrabarra literal, ela deve ser escapada na expressão regular. Com a notação de string bruta, isso significa `r"\\"`. Sem a notação de string bruta, deve-se usar `"\\\\"`, tornando as seguintes linhas de código funcionalmente idênticas:

```
>>> re.match(r"\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
>>> re.match("\\\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
```

Escrevendo um tokenizador

Um tokenizador, tokenizer ou scanner analisa uma string para categorizar grupos de caracteres. Este é um primeiro passo útil para escrever um compilador ou interpretador.

As categorias de texto são especificadas com expressões regulares. A técnica é combiná-las em uma única expressão regular mestre e fazer um loop em correspondências sucessivas:

```
import collections
import re

Token = collections.namedtuple('Token', ['type', 'value', 'line', 'column'])

def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('ASSIGN',   r':='),           # Assignment operator
        ('END',      r';'),            # Statement terminator
        ('ID',       r'[A-Za-z]+'),    # Identifiers
        ('OP',       r'[+\-*/]'),      # Arithmetic operators
        ('NEWLINE',  r'\n'),           # Line endings
        ('SKIP',     r'[ \t]+'),       # Skip over spaces and tabs
        ('MISMATCH', r'.'),            # Any other character
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        column = mo.start() - line_start
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'ID' and value in keywords:
            kind = value
        elif kind == 'NEWLINE':
            line_start = mo.end()
            line_num += 1
            continue
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise RuntimeError(f'{value!r} unexpected on line {line_num}')
        yield Token(kind, value, line_num, column)

statements = '''
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
```

(continua na próxima página)

(continuação da página anterior)

```
'''

for token in tokenize(statements):
    print(token)
```

O tokenizador produz a seguinte saída:

```
Token(type='IF', value='IF', line=2, column=4)
Token(type='ID', value='quantity', line=2, column=7)
Token(type='THEN', value='THEN', line=2, column=16)
Token(type='ID', value='total', line=3, column=8)
Token(type='ASSIGN', value=':=', line=3, column=14)
Token(type='ID', value='total', line=3, column=17)
Token(type='OP', value='+', line=3, column=23)
Token(type='ID', value='price', line=3, column=25)
Token(type='OP', value='*', line=3, column=31)
Token(type='ID', value='quantity', line=3, column=33)
Token(type='END', value=';', line=3, column=41)
Token(type='ID', value='tax', line=4, column=8)
Token(type='ASSIGN', value=':=', line=4, column=12)
Token(type='ID', value='price', line=4, column=15)
Token(type='OP', value='*', line=4, column=21)
Token(type='NUMBER', value=0.05, line=4, column=23)
Token(type='END', value=';', line=4, column=27)
Token(type='ENDIF', value='ENDIF', line=5, column=4)
Token(type='END', value=';', line=5, column=9)
```

6.3 `difflib` — Helpers for computing deltas

Código Fonte: Lib/difflib.py

This module provides classes and functions for comparing sequences. It can be used for example, for comparing files, and can produce difference information in various formats, including HTML and context and unified diffs. For comparing directories and files, see also, the *filecmp* module.

class difflib.**SequenceMatcher**

This is a flexible class for comparing pairs of sequences of any type, so long as the sequence elements are *hashable*. The basic algorithm predates, and is a little fancier than, an algorithm published in the late 1980's by Ratcliff and Obershelp under the hyperbolic name "gestalt pattern matching." The idea is to find the longest contiguous matching subsequence that contains no "junk" elements; these "junk" elements are ones that are uninteresting in some sense, such as blank lines or whitespace. (Handling junk is an extension to the Ratcliff and Obershelp algorithm.) The same idea is then applied recursively to the pieces of the sequences to the left and to the right of the matching subsequence. This does not yield minimal edit sequences, but does tend to yield matches that "look right" to people.

Timing: The basic Ratcliff-Obershelp algorithm is cubic time in the worst case and quadratic time in the expected case. *SequenceMatcher* is quadratic time for the worst case and has expected-case behavior dependent in a complicated way on how many elements the sequences have in common; best case time is linear.

Automatic junk heuristic: *SequenceMatcher* supports a heuristic that automatically treats certain sequence items as junk. The heuristic counts how many times each individual item appears in the sequence. If an item's duplicates (after the first one) account for more than 1% of the sequence and the sequence is at least 200 items

long, this item is marked as "popular" and is treated as junk for the purpose of sequence matching. This heuristic can be turned off by setting the `autojunk` argument to `False` when creating the *SequenceMatcher*.

Novo na versão 3.2: The *autojunk* parameter.

class difflib.**Differ**

This is a class for comparing sequences of lines of text, and producing human-readable differences or deltas. Differ uses *SequenceMatcher* both to compare sequences of lines, and to compare sequences of characters within similar (near-matching) lines.

Each line of a *Differ* delta begins with a two-letter code:

Código	Significado
`'- '`	line unique to sequence 1
`'+ '`	line unique to sequence 2
`'  '`	line common to both sequences
`'? '`	line not present in either input sequence

Lines beginning with '`?`' attempt to guide the eye to intraline differences, and were not present in either input sequence. These lines can be confusing if the sequences contain tab characters.

class difflib.**HtmlDiff**

This class can be used to create an HTML table (or a complete HTML file containing the table) showing a side by side, line by line comparison of text with inter-line and intra-line change highlights. The table can be generated in either full or contextual difference mode.

The constructor for this class is:

__init__(*tabsize=8*, *wrapcolumn=None*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

Initializes instance of *HtmlDiff*.

tabsize is an optional keyword argument to specify tab stop spacing and defaults to `8`.

wrapcolumn is an optional keyword to specify column number where lines are broken and wrapped, defaults to `None` where lines are not wrapped.

linejunk and *charjunk* are optional keyword arguments passed into *ndiff()* (used by *HtmlDiff* to generate the side by side HTML differences). See *ndiff()* documentation for argument default values and descriptions.

The following methods are public:

make_file(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*, ***, *charset='utf-8'*)

Compares *fromlines* and *tolines* (lists of strings) and returns a string which is a complete HTML file containing a table showing line by line differences with inter-line and intra-line changes highlighted.

fromdesc and *todesc* are optional keyword arguments to specify from/to file column header strings (both default to an empty string).

context and *numlines* are both optional keyword arguments. Set *context* to `True` when contextual differences are to be shown, else the default is `False` to show the full files. *numlines* defaults to `5`. When *context* is `True` *numlines* controls the number of context lines which surround the difference highlights. When *context* is `False` *numlines* controls the number of lines which are shown before a difference highlight when using the "next" hyperlinks (setting to zero would cause the "next" hyperlinks to place the next difference highlight at the top of the browser without any leading context).

Alterado na versão 3.5: *charset* keyword-only argument was added. The default charset of HTML document changed from `'ISO-8859-1'` to `'utf-8'`.

make_table(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*)
Compares *fromlines* and *tolines* (lists of strings) and returns a string which is a complete HTML table showing line by line differences with inter-line and intra-line changes highlighted.

The arguments for this method are the same as those for the *make_file()* method.

`Tools/scripts/diff.py` is a command-line front-end to this class and contains a good example of its use.

difflib.**context_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)
Compare *a* and *b* (lists of strings); return a delta (a *generator* generating the delta lines) in context diff format.

Context diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in a before/after style. The number of context lines is set by *n* which defaults to three.

By default, the diff control lines (those with `***` or `---`) are created with a trailing newline. This is helpful so that inputs created from *io.IOBase.readlines()* result in diffs that are suitable for use with *io.IOBase.writelines()* since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the *lineterm* argument to `""` so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for *fromfile*, *tofile*, *fromfiledate*, and *tofiledate*. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(context_diff(s1, s2, fromfile='before.py', tofile=
↪'after.py'))
*** before.py
--- after.py
***************
*** 1,4 ****
! bacon
! eggs
! ham
  guido
--- 1,4 ----
! python
! eggy
! hamster
  guido
```

See *A command-line interface to difflib* for a more detailed example.

difflib.**get_close_matches**(*word*, *possibilities*, *n=3*, *cutoff=0.6*)
Return a list of the best "good enough" matches. *word* is a sequence for which close matches are desired (typically a string), and *possibilities* is a list of sequences against which to match *word* (typically a list of strings).

Optional argument *n* (default `3`) is the maximum number of close matches to return; *n* must be greater than `0`.

Optional argument *cutoff* (default `0.6`) is a float in the range [0, 1]. Possibilities that don't score at least that similar to *word* are ignored.

The best (no more than *n*) matches among the possibilities are returned in a list, sorted by similarity score, most similar first.

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
```

(continua na próxima página)

(continuação da página anterior)

```
['while']
>>> get_close_matches('pineapple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

difflib.**ndiff**(*a*, *b*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

Compare *a* and *b* (lists of strings); return a *Differ*-style delta (a *generator* generating the delta lines).

Optional keyword parameters *linejunk* and *charjunk* are filtering functions (or `None`):

linejunk: A function that accepts a single string argument, and returns true if the string is junk, or false if not. The default is `None`. There is also a module-level function *IS_LINE_JUNK()*, which filters out lines without visible characters, except for at most one pound character (`'#'`) – however the underlying *SequenceMatcher* class does a dynamic analysis of which lines are so frequent as to constitute noise, and this usually works better than using this function.

charjunk: A function that accepts a character (a string of length 1), and returns if the character is junk, or false if not. The default is module-level function *IS_CHARACTER_JUNK()*, which filters out whitespace characters (a blank or tab; it's a bad idea to include newline in this!).

`Tools/scripts/ndiff.py` is a command-line front-end to this function.

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> print(''.join(diff), end="")
- one
?  ^
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

difflib.**restore**(*sequence*, *which*)

Return one of the two sequences that generated a delta.

Given a *sequence* produced by *Differ.compare()* or *ndiff()*, extract lines originating from file 1 or 2 (parameter *which*), stripping off line prefixes.

Exemplo:

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print(''.join(restore(diff, 1)), end="")
one
two
three
>>> print(''.join(restore(diff, 2)), end="")
ore
tree
emu
```

difflib.**unified_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

Compare *a* and *b* (lists of strings); return a delta (a *generator* generating the delta lines) in unified diff format.

Unified diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in an inline style (instead of separate before/after blocks). The number of context lines is set by *n* which defaults to three.

By default, the diff control lines (those with `---`, `+++`, or `@@`) are created with a trailing newline. This is helpful so that inputs created from *io.IOBase.readlines()* result in diffs that are suitable for use with *io.IOBase.writelines()* since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the *lineterm* argument to `""` so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for *fromfile*, *tofile*, *fromfiledate*, and *tofiledate*. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(unified_diff(s1, s2, fromfile='before.py', tofile=
↪'after.py'))
--- before.py
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
+eggy
+hamster
 guido
```

See *A command-line interface to difflib* for a more detailed example.

`difflib.`**`diff_bytes`**(*dfunc*, *a*, *b*, *fromfile=b''*, *tofile=b''*, *fromfiledate=b''*, *tofiledate=b''*, *n=3*, *lineterm=b'\n'*)

Compare *a* and *b* (lists of bytes objects) using *dfunc*; yield a sequence of delta lines (also bytes) in the format returned by *dfunc*. *dfunc* must be a callable, typically either *unified_diff()* or *context_diff()*.

Allows you to compare data with unknown or inconsistent encoding. All inputs except *n* must be bytes objects, not str. Works by losslessly converting all inputs (except *n*) to str, and calling `dfunc(a, b, fromfile, tofile, fromfiledate, tofiledate, n, lineterm)`. The output of *dfunc* is then converted back to bytes, so the delta lines that you receive have the same unknown/inconsistent encodings as *a* and *b*.

Novo na versão 3.5.

`difflib.`**`IS_LINE_JUNK`**(*line*)

Return `True` for ignorable lines. The line *line* is ignorable if *line* is blank or contains a single `'#'`, otherwise it is not ignorable. Used as a default for parameter *linejunk* in *ndiff()* in older versions.

`difflib.`**`IS_CHARACTER_JUNK`**(*ch*)

Return `True` for ignorable characters. The character *ch* is ignorable if *ch* is a space or tab, otherwise it is not ignorable. Used as a default for parameter *charjunk* in *ndiff()*.

Ver também:

Pattern Matching: The Gestalt Approach Discussion of a similar algorithm by John W. Ratcliff and D. E. Metzener. This was published in Dr. Dobb's Journal in July, 1988.

6.3.1 SequenceMatcher Objects

The *SequenceMatcher* class has this constructor:

class difflib.**SequenceMatcher**(*isjunk=None*, *a=''*, *b=''*, *autojunk=True*)

Optional argument *isjunk* must be `None` (the default) or a one-argument function that takes a sequence element and returns true if and only if the element is "junk" and should be ignored. Passing `None` for *isjunk* is equivalent to passing `lambda x: False`; in other words, no elements are ignored. For example, pass:

```
lambda x: x in " \t"
```

if you're comparing lines as sequences of characters, and don't want to synch up on blanks or hard tabs.

The optional arguments *a* and *b* are sequences to be compared; both default to empty strings. The elements of both sequences must be *hashable*.

The optional argument *autojunk* can be used to disable the automatic junk heuristic.

Novo na versão 3.2: The *autojunk* parameter.

SequenceMatcher objects get three data attributes: *bjunk* is the set of elements of *b* for which *isjunk* is `True`; *bpopular* is the set of non-junk elements considered popular by the heuristic (if it is not disabled); *b2j* is a dict mapping the remaining elements of *b* to a list of positions where they occur. All three are reset whenever *b* is reset with *set_seqs()* or *set_seq2()*.

Novo na versão 3.2: The *bjunk* and *bpopular* attributes.

SequenceMatcher objects have the following methods:

set_seqs(*a*, *b*)

Set the two sequences to be compared.

SequenceMatcher computes and caches detailed information about the second sequence, so if you want to compare one sequence against many sequences, use *set_seq2()* to set the commonly used sequence once and call *set_seq1()* repeatedly, once for each of the other sequences.

set_seq1(*a*)

Set the first sequence to be compared. The second sequence to be compared is not changed.

set_seq2(*b*)

Set the second sequence to be compared. The first sequence to be compared is not changed.

find_longest_match(*alo*, *ahi*, *blo*, *bhi*)

Find longest matching block in `a[alo:ahi]` and `b[blo:bhi]`.

If *isjunk* was omitted or `None`, *find_longest_match()* returns `(i, j, k)` such that `a[i:i+k]` is equal to `b[j:j+k]`, where `alo <= i <= i+k <= ahi` and `blo <= j <= j+k <= bhi`. For all `(i', j', k')` meeting those conditions, the additional conditions `k >= k'`, `i <= i'`, and if `i == i'`, `j <= j'` are also met. In other words, of all maximal matching blocks, return one that starts earliest in *a*, and of all those maximal matching blocks that start earliest in *a*, return the one that starts earliest in *b*.

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

If *isjunk* was provided, first the longest matching block is determined as above, but with the additional restriction that no junk element appears in the block. Then that block is extended as far as possible by matching (only) junk elements on both sides. So the resulting block never matches on junk except as identical junk happens to be adjacent to an interesting match.

Here's the same example as before, but considering blanks to be junk. That prevents `' abcd'` from matching the `' abcd'` at the tail end of the second sequence directly. Instead only the `'abcd'` can match, and matches the leftmost `'abcd'` in the second sequence:

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

If no blocks match, this returns `(alo, blo, 0)`.

This method returns a *named tuple* `Match(a, b, size)`.

get_matching_blocks()

Return list of triples describing non-overlapping matching subsequences. Each triple is of the form `(i, j, n)`, and means that `a[i:i+n] == b[j:j+n]`. The triples are monotonically increasing in *i* and *j*.

The last triple is a dummy, and has the value `(len(a), len(b), 0)`. It is the only triple with `n == 0`. If `(i, j, n)` and `(i', j', n')` are adjacent triples in the list, and the second is not the last triple in the list, then `i+n < i'` or `j+n < j'`; in other words, adjacent triples always describe non-adjacent equal blocks.

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

get_opcodes()

Return list of 5-tuples describing how to turn *a* into *b*. Each tuple is of the form `(tag, i1, i2, j1, j2)`. The first tuple has `i1 == j1 == 0`, and remaining tuples have *i1* equal to the *i2* from the preceding tuple, and, likewise, *j1* equal to the previous *j2*.

The *tag* values are strings, with these meanings:

Valor	Significado
`'replace'`	`a[i1:i2]` should be replaced by `b[j1:j2]`.
`'delete'`	`a[i1:i2]` should be deleted. Note that `j1 == j2` in this case.
`'insert'`	`b[j1:j2]` should be inserted at `a[i1:i1]`. Note that `i1 == i2` in this case.
`'equal'`	`a[i1:i2] == b[j1:j2]` (the sub-sequences are equal).

Por exemplo:

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print('{:7}   a[{}:{}] --> b[{}:{}] {!r:>8} --> {!r}'.format(
...         tag, i1, i2, j1, j2, a[i1:i2], b[j1:j2]))
delete    a[0:1] --> b[0:0]      'q' --> ''
equal     a[1:3] --> b[0:2]     'ab' --> 'ab'
replace   a[3:4] --> b[2:3]      'x' --> 'y'
equal     a[4:6] --> b[3:5]     'cd' --> 'cd'
insert    a[6:6] --> b[5:6]       '' --> 'f'
```

get_grouped_opcodes(*n=3*)

Return a *generator* of groups with up to *n* lines of context.

Starting with the groups returned by *get_opcodes()*, this method splits out smaller change clusters and eliminates intervening ranges which have no changes.

The groups are returned in the same format as *get_opcodes()*.

ratio()
Return a measure of the sequences' similarity as a float in the range [0, 1].

Where T is the total number of elements in both sequences, and M is the number of matches, this is 2.0*M / T. Note that this is `1.0` if the sequences are identical, and `0.0` if they have nothing in common.

This is expensive to compute if *get_matching_blocks()* or *get_opcodes()* hasn't already been called, in which case you may want to try *quick_ratio()* or *real_quick_ratio()* first to get an upper bound.

Nota: Caution: The result of a *ratio()* call may depend on the order of the arguments. For instance:

```
>>> SequenceMatcher(None, 'tide', 'diet').ratio()
0.25
>>> SequenceMatcher(None, 'diet', 'tide').ratio()
0.5
```

---

quick_ratio()
Return an upper bound on *ratio()* relatively quickly.

real_quick_ratio()
Return an upper bound on *ratio()* very quickly.

The three methods that return the ratio of matching to total characters can give different results due to differing levels of approximation, although `quick_ratio()` and `real_quick_ratio()` are always at least as large as `ratio()`:

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

6.3.2 SequenceMatcher Examples

This example compares two strings, considering blanks to be "junk":

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                     "private Thread currentThread;",
...                     "private volatile Thread currentThread;")
```

`ratio()` returns a float in [0, 1], measuring the similarity of the sequences. As a rule of thumb, a `ratio()` value over 0.6 means the sequences are close matches:

```
>>> print(round(s.ratio(), 3))
0.866
```

If you're only interested in where the sequences match, `get_matching_blocks()` is handy:

```
>>> for block in s.get_matching_blocks():
...     print("a[%d] and b[%d] match for %d elements" % block)
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

Note that the last tuple returned by `get_matching_blocks()` is always a dummy, `(len(a), len(b), 0)`, and this is the only case in which the last tuple element (number of elements matched) is `0`.

If you want to know how to change the first sequence into the second, use `get_opcodes()`:

```
>>> for opcode in s.get_opcodes():
...     print("%6s a[%d:%d] b[%d:%d]" % opcode)
 equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
 equal a[8:29] b[17:38]
```

Ver também:

- The *get_close_matches()* function in this module which shows how simple code building on *SequenceMatcher* can be used to do useful work.
- Simple version control recipe for a small application built with *SequenceMatcher*.

6.3.3 Differ Objects

Note that *Differ*-generated deltas make no claim to be **minimal** diffs. To the contrary, minimal diffs are often counter-intuitive, because they synch up anywhere possible, sometimes accidental matches 100 pages apart. Restricting synch points to contiguous matches preserves some notion of locality, at the occasional cost of producing a longer diff.

The *Differ* class has this constructor:

class difflib.**Differ**(*linejunk=None*, *charjunk=None*)

Optional keyword parameters *linejunk* and *charjunk* are for filter functions (or `None`):

linejunk: A function that accepts a single string argument, and returns true if the string is junk. The default is `None`, meaning that no line is considered junk.

charjunk: A function that accepts a single character argument (a string of length 1), and returns true if the character is junk. The default is `None`, meaning that no character is considered junk.

These junk-filtering functions speed up matching to find differences and do not cause any differing lines or characters to be ignored. Read the description of the *find_longest_match()* method's *isjunk* parameter for an explanation.

Differ objects are used (deltas generated) via a single method:

compare(*a*, *b*)

Compare two sequences of lines, and generate the delta (a sequence of lines).

Each sequence must contain individual single-line strings ending with newlines. Such sequences can be obtained from the *readlines()* method of file-like objects. The delta generated also consists of newline-terminated strings, ready to be printed as-is via the *writelines()* method of a file-like object.

6.3.4 Differ Example

This example compares two texts. First we set up the texts, sequences of individual single-line strings ending with newlines (such sequences can also be obtained from the `readlines()` method of file-like objects):

```
>>> text1 = '''  1. Beautiful is better than ugly.
...   2. Explicit is better than implicit.
...   3. Simple is better than complex.
...   4. Complex is better than complicated.
... '''.splitlines(keepends=True)
```

(continua na próxima página)

(continuação da página anterior)

```
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = '''  1. Beautiful is better than ugly.
...   3.   Simple is better than complex.
...   4. Complicated is better than complex.
...   5. Flat is better than nested.
... '''.splitlines(keepends=True)
```

Next we instantiate a Differ object:

```
>>> d = Differ()
```

Note that when instantiating a *Differ* object we may pass functions to filter out line and character "junk." See the *Differ()* constructor for details.

Finally, we compare the two:

```
>>> result = list(d.compare(text1, text2))
```

`result` is a list of strings, so let's pretty-print it:

```
>>> from pprint import pprint
>>> pprint(result)
['    1. Beautiful is better than ugly.\n',
 '-   2. Explicit is better than implicit.\n',
 '-   3. Simple is better than complex.\n',
 '+   3.   Simple is better than complex.\n',
 '?     ++\n',
 '-   4. Complex is better than complicated.\n',
 '?            ^                     ---- ^\n',
 '+   4. Complicated is better than complex.\n',
 '?           +++ ^                         ^\n',
 '+   5. Flat is better than nested.\n']
```

As a single multi-line string it looks like this:

```
>>> import sys
>>> sys.stdout.writelines(result)
    1. Beautiful is better than ugly.
-   2. Explicit is better than implicit.
-   3. Simple is better than complex.
+   3.   Simple is better than complex.
?     ++
-   4. Complex is better than complicated.
?            ^                     ---- ^
+   4. Complicated is better than complex.
?           +++ ^                         ^
+   5. Flat is better than nested.
```

6.3.5 A command-line interface to difflib

This example shows how to use difflib to create a `diff`-like utility. It is also contained in the Python source distribution, as `Tools/scripts/diff.py`.

```
#!/usr/bin/env python3
""" Command line interface to difflib.py providing diffs in four formats:

* ndiff:    lists every line and highlights interline changes.
* context:  highlights clusters of changes in a before/after format.
* unified:  highlights clusters of changes in an inline format.
* html:     generates side by side comparison with change highlights.

"""

import sys, os, difflib, argparse
from datetime import datetime, timezone

def file_mtime(path):
    t = datetime.fromtimestamp(os.stat(path).st_mtime,
                               timezone.utc)
    return t.astimezone().isoformat()

def main():

    parser = argparse.ArgumentParser()
    parser.add_argument('-c', action='store_true', default=False,
                        help='Produce a context format diff (default)')
    parser.add_argument('-u', action='store_true', default=False,
                        help='Produce a unified format diff')
    parser.add_argument('-m', action='store_true', default=False,
                        help='Produce HTML side by side diff '
                             '(can use -c and -l in conjunction)')
    parser.add_argument('-n', action='store_true', default=False,
                        help='Produce a ndiff format diff')
    parser.add_argument('-l', '--lines', type=int, default=3,
                        help='Set number of context lines (default 3)')
    parser.add_argument('fromfile')
    parser.add_argument('tofile')
    options = parser.parse_args()

    n = options.lines
    fromfile = options.fromfile
    tofile = options.tofile

    fromdate = file_mtime(fromfile)
    todate = file_mtime(tofile)
    with open(fromfile) as ff:
        fromlines = ff.readlines()
    with open(tofile) as tf:
        tolines = tf.readlines()

    if options.u:
        diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile, fromdate,␣
↪todate, n=n)
    elif options.n:
        diff = difflib.ndiff(fromlines, tolines)
    elif options.m:
```

(continua na próxima página)

(continuação da página anterior)

```
        diff = difflib.HtmlDiff().make_file(fromlines,tolines,fromfile,tofile,
↪context=options.c,numlines=n)
    else:
        diff = difflib.context_diff(fromlines, tolines, fromfile, tofile, fromdate,␣
↪todate, n=n)

    sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()
```

6.4 `textwrap` — Text wrapping and filling

Código Fonte: Lib/textwrap.py

The *textwrap* module provides some convenience functions, as well as *TextWrapper*, the class that does all the work. If you're just wrapping or filling one or two text strings, the convenience functions should be good enough; otherwise, you should use an instance of *TextWrapper* for efficiency.

`textwrap.`**`wrap`**(*text*, *width=70*, ***kwargs*)

Wraps the single paragraph in *text* (a string) so every line is at most *width* characters long. Returns a list of output lines, without final newlines.

Optional keyword arguments correspond to the instance attributes of *TextWrapper*, documented below. *width* defaults to `70`.

See the *TextWrapper.wrap()* method for additional details on how *wrap()* behaves.

`textwrap.`**`fill`**(*text*, *width=70*, ***kwargs*)

Wraps the single paragraph in *text*, and returns a single string containing the wrapped paragraph. *fill()* is shorthand for

```
"\n".join(wrap(text, ...))
```

In particular, *fill()* accepts exactly the same keyword arguments as *wrap()*.

`textwrap.`**`shorten`**(*text*, *width*, ***kwargs*)

Collapse and truncate the given *text* to fit in the given *width*.

First the whitespace in *text* is collapsed (all whitespace is replaced by single spaces). If the result fits in the *width*, it is returned. Otherwise, enough words are dropped from the end so that the remaining words plus the `placeholder` fit within `width`:

```
>>> textwrap.shorten("Hello  world!", width=12)
'Hello world!'
>>> textwrap.shorten("Hello  world!", width=11)
'Hello [...]'
>>> textwrap.shorten("Hello world", width=10, placeholder="...")
'Hello...'
```

Optional keyword arguments correspond to the instance attributes of *TextWrapper*, documented below. Note that the whitespace is collapsed before the text is passed to the *TextWrapper* *fill()* function, so changing the value of *tabsize*, *expand_tabs*, *drop_whitespace*, and *replace_whitespace* will have no effect.

Novo na versão 3.4.

textwrap.**dedent**(*text*)
Remove any common leading whitespace from every line in *text*.

This can be used to make triple-quoted strings line up with the left edge of the display, while still presenting them in the source code in indented form.

Note that tabs and spaces are both treated as whitespace, but they are not equal: the lines `"  hello"` and `"\thello"` are considered to have no common leading whitespace.

Lines containing only whitespace are ignored in the input and normalized to a single newline character in the output.

Por exemplo:

```
def test():
    # end first line with \ to avoid the empty line!
    s = '''\
    hello
      world
    '''
    print(repr(s))          # prints '    hello\n      world\n    '
    print(repr(dedent(s)))  # prints 'hello\n  world\n'
```

textwrap.**indent**(*text*, *prefix*, *predicate=None*)
Add *prefix* to the beginning of selected lines in *text*.

Lines are separated by calling `text.splitlines(True)`.

By default, *prefix* is added to all lines that do not consist solely of whitespace (including any line endings).

Por exemplo:

```
>>> s = 'hello\n\n \nworld'
>>> indent(s, '  ')
'  hello\n\n \n  world'
```

The optional *predicate* argument can be used to control which lines are indented. For example, it is easy to add *prefix* to even empty and whitespace-only lines:

```
>>> print(indent(s, '+ ', lambda line: True))
+ hello
+
+
+ world
```

Novo na versão 3.3.

wrap(), *fill()* and *shorten()* work by creating a *TextWrapper* instance and calling a single method on it. That instance is not reused, so for applications that process many text strings using *wrap()* and/or *fill()*, it may be more efficient to create your own *TextWrapper* object.

Text is preferably wrapped on whitespaces and right after the hyphens in hyphenated words; only then will long words be broken if necessary, unless *TextWrapper.break_long_words* is set to false.

class textwrap.**TextWrapper**(***kwargs*)
The *TextWrapper* constructor accepts a number of optional keyword arguments. Each keyword argument corresponds to an instance attribute, so for example

```
wrapper = TextWrapper(initial_indent="* ")
```

é o mesmo que

```
wrapper = TextWrapper()
wrapper.initial_indent = "* "
```

You can re-use the same *TextWrapper* object many times, and you can change any of its options through direct assignment to instance attributes between uses.

The *TextWrapper* instance attributes (and keyword arguments to the constructor) are as follows:

width
(default: `70`) The maximum length of wrapped lines. As long as there are no individual words in the input text longer than *width*, *TextWrapper* guarantees that no output line will be longer than *width* characters.

expand_tabs
(default: `True`) If true, then all tab characters in *text* will be expanded to spaces using the `expandtabs()` method of *text*.

tabsize
(default: `8`) If *expand_tabs* is true, then all tab characters in *text* will be expanded to zero or more spaces, depending on the current column and the given tab size.

Novo na versão 3.3.

replace_whitespace
(default: `True`) If true, after tab expansion but before wrapping, the *wrap()* method will replace each whitespace character with a single space. The whitespace characters replaced are as follows: tab, newline, vertical tab, formfeed, and carriage return (`'\t\n\v\f\r'`).

Nota: If *expand_tabs* is false and *replace_whitespace* is true, each tab character will be replaced by a single space, which is *not* the same as tab expansion.

Nota: If *replace_whitespace* is false, newlines may appear in the middle of a line and cause strange output. For this reason, text should be split into paragraphs (using *str.splitlines()* or similar) which are wrapped separately.

---

drop_whitespace
(default: `True`) If true, whitespace at the beginning and ending of every line (after wrapping but before indenting) is dropped. Whitespace at the beginning of the paragraph, however, is not dropped if non-whitespace follows it. If whitespace being dropped takes up an entire line, the whole line is dropped.

initial_indent
(default: `''`) String that will be prepended to the first line of wrapped output. Counts towards the length of the first line. The empty string is not indented.

subsequent_indent
(default: `''`) String that will be prepended to all lines of wrapped output except the first. Counts towards the length of each line except the first.

fix_sentence_endings
(default: `False`) If true, *TextWrapper* attempts to detect sentence endings and ensure that sentences are always separated by exactly two spaces. This is generally desired for text in a monospaced font. However, the sentence detection algorithm is imperfect: it assumes that a sentence ending consists of a lowercase letter followed by one of `'.'`, `'!'`, or `'?'`, possibly followed by one of `'"'` or `"'"`, followed by a space. One problem with this is algorithm is that it is unable to detect the difference between “Dr.” in

```
[...] Dr. Frankenstein's monster [...]
```

e “Spot.” em

```
[...] See Spot. See Spot run [...]
```

fix_sentence_endings is false by default.

Since the sentence detection algorithm relies on `string.lowercase` for the definition of “lowercase letter”, and a convention of using two spaces after a period to separate sentences on the same line, it is specific to English-language texts.

break_long_words
(default: `True`) If true, then words longer than *width* will be broken in order to ensure that no lines are longer than *width*. If it is false, long words will not be broken, and some lines may be longer than *width*. (Long words will be put on a line by themselves, in order to minimize the amount by which *width* is exceeded.)

break_on_hyphens
(default: `True`) If true, wrapping will occur preferably on whitespaces and right after hyphens in compound words, as it is customary in English. If false, only whitespaces will be considered as potentially good places for line breaks, but you need to set *break_long_words* to false if you want truly insecable words. Default behaviour in previous versions was to always allow breaking hyphenated words.

max_lines
(default: `None`) If not `None`, then the output will contain at most *max_lines* lines, with *placeholder* appearing at the end of the output.

Novo na versão 3.4.

placeholder
(default: `' [...]'`) String that will appear at the end of the output text if it has been truncated.

Novo na versão 3.4.

TextWrapper also provides some public methods, analogous to the module-level convenience functions:

wrap(*text*)
Wraps the single paragraph in *text* (a string) so every line is at most *width* characters long. All wrapping options are taken from instance attributes of the *TextWrapper* instance. Returns a list of output lines, without final newlines. If the wrapped output has no content, the returned list is empty.

fill(*text*)
Wraps the single paragraph in *text*, and returns a single string containing the wrapped paragraph.

6.5 `unicodedata` — Unicode Database

This module provides access to the Unicode Character Database (UCD) which defines character properties for all Unicode characters. The data contained in this database is compiled from the UCD version 11.0.0.

The module uses the same names and symbols as defined by Unicode Standard Annex #44, “Unicode Character Database”. It defines the following functions:

`unicodedata.`**lookup**(*name*)
Look up character by name. If a character with the given name is found, return the corresponding character. If not found, *KeyError* is raised.

Alterado na versão 3.3: Support for name aliases[1] and named sequences[2] has been added.

[1] http://www.unicode.org/Public/11.0.0/ucd/NameAliases.txt
[2] http://www.unicode.org/Public/11.0.0/ucd/NamedSequences.txt

unicodedata.**name**(*chr*[, *default*])
Returns the name assigned to the character *chr* as a string. If no name is defined, *default* is returned, or, if not given, `ValueError` is raised.

unicodedata.**decimal**(*chr*[, *default*])
Returns the decimal value assigned to the character *chr* as integer. If no such value is defined, *default* is returned, or, if not given, `ValueError` is raised.

unicodedata.**digit**(*chr*[, *default*])
Returns the digit value assigned to the character *chr* as integer. If no such value is defined, *default* is returned, or, if not given, `ValueError` is raised.

unicodedata.**numeric**(*chr*[, *default*])
Returns the numeric value assigned to the character *chr* as float. If no such value is defined, *default* is returned, or, if not given, `ValueError` is raised.

unicodedata.**category**(*chr*)
Returns the general category assigned to the character *chr* as string.

unicodedata.**bidirectional**(*chr*)
Returns the bidirectional class assigned to the character *chr* as string. If no such value is defined, an empty string is returned.

unicodedata.**combining**(*chr*)
Returns the canonical combining class assigned to the character *chr* as integer. Returns `0` if no combining class is defined.

unicodedata.**east_asian_width**(*chr*)
Returns the east asian width assigned to the character *chr* as string.

unicodedata.**mirrored**(*chr*)
Returns the mirrored property assigned to the character *chr* as integer. Returns `1` if the character has been identified as a "mirrored" character in bidirectional text, `0` otherwise.

unicodedata.**decomposition**(*chr*)
Returns the character decomposition mapping assigned to the character *chr* as string. An empty string is returned in case no such mapping is defined.

unicodedata.**normalize**(*form*, *unistr*)
Return the normal form *form* for the Unicode string *unistr*. Valid values for *form* are 'NFC', 'NFKC', 'NFD', and 'NFKD'.

The Unicode standard defines various normalization forms of a Unicode string, based on the definition of canonical equivalence and compatibility equivalence. In Unicode, several characters can be expressed in various way. For example, the character U+00C7 (LATIN CAPITAL LETTER C WITH CEDILLA) can also be expressed as the sequence U+0043 (LATIN CAPITAL LETTER C) U+0327 (COMBINING CEDILLA).

For each character, there are two normal forms: normal form C and normal form D. Normal form D (NFD) is also known as canonical decomposition, and translates each character into its decomposed form. Normal form C (NFC) first applies a canonical decomposition, then composes pre-combined characters again.

In addition to these two forms, there are two additional normal forms based on compatibility equivalence. In Unicode, certain characters are supported which normally would be unified with other characters. For example, U+2160 (ROMAN NUMERAL ONE) is really the same thing as U+0049 (LATIN CAPITAL LETTER I). However, it is supported in Unicode for compatibility with existing character sets (e.g. gb2312).

The normal form KD (NFKD) will apply the compatibility decomposition, i.e. replace all compatibility characters with their equivalents. The normal form KC (NFKC) first applies the compatibility decomposition, followed by the canonical composition.

Even if two unicode strings are normalized and look the same to a human reader, if one has combining characters and the other doesn't, they may not compare equal.

In addition, the module exposes the following constant:

unicodedata.**unidata_version**
The version of the Unicode database used in this module.

unicodedata.**ucd_3_2_0**
This is an object that has the same methods as the entire module, but uses the Unicode database version 3.2 instead, for applications that require this specific version of the Unicode database (such as IDNA).

Exemplos:

```
>>> import unicodedata
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'{'
>>> unicodedata.name('/')
'SOLIDUS'
>>> unicodedata.decimal('9')
9
>>> unicodedata.decimal('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not a decimal
>>> unicodedata.category('A')  # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional('\u0660') # 'A'rabic, 'N'umber
'AN'
```

6.6 stringprep — Internet String Preparation

Código Fonte: Lib/stringprep.py

When identifying things (such as host names) in the internet, it is often necessary to compare such identifications for "equality". Exactly how this comparison is executed may depend on the application domain, e.g. whether it should be case-insensitive or not. It may be also necessary to restrict the possible identifications, to allow only identifications consisting of "printable" characters.

RFC 3454 defines a procedure for "preparing" Unicode strings in internet protocols. Before passing strings onto the wire, they are processed with the preparation procedure, after which they have a certain normalized form. The RFC defines a set of tables, which can be combined into profiles. Each profile must define which tables it uses, and what other optional parts of the stringprep procedure are part of the profile. One example of a stringprep profile is nameprep, which is used for internationalized domain names.

The module *stringprep* only exposes the tables from RFC 3454. As these tables would be very large to represent them as dictionaries or lists, the module uses the Unicode character database internally. The module source code itself was generated using the mkstringprep.py utility.

As a result, these tables are exposed as functions, not as data structures. There are two kinds of tables in the RFC: sets and mappings. For a set, *stringprep* provides the "characteristic function", i.e. a function that returns True if the parameter is part of the set. For mappings, it provides the mapping function: given the key, it returns the associated value. Below is a list of all functions available in the module.

stringprep.**in_table_a1**(*code*)
Determine whether *code* is in tableA.1 (Unassigned code points in Unicode 3.2).

stringprep.**in_table_b1**(*code*)
Determine whether *code* is in tableB.1 (Commonly mapped to nothing).

stringprep.**map_table_b2**(*code*)
Return the mapped value for *code* according to tableB.2 (Mapping for case-folding used with NFKC).

stringprep.**map_table_b3**(*code*)
Return the mapped value for *code* according to tableB.3 (Mapping for case-folding used with no normalization).

stringprep.**in_table_c11**(*code*)
Determine whether *code* is in tableC.1.1 (ASCII space characters).

stringprep.**in_table_c12**(*code*)
Determine whether *code* is in tableC.1.2 (Non-ASCII space characters).

stringprep.**in_table_c11_c12**(*code*)
Determine whether *code* is in tableC.1 (Space characters, union of C.1.1 and C.1.2).

stringprep.**in_table_c21**(*code*)
Determine whether *code* is in tableC.2.1 (ASCII control characters).

stringprep.**in_table_c22**(*code*)
Determine whether *code* is in tableC.2.2 (Non-ASCII control characters).

stringprep.**in_table_c21_c22**(*code*)
Determine whether *code* is in tableC.2 (Control characters, union of C.2.1 and C.2.2).

stringprep.**in_table_c3**(*code*)
Determine whether *code* is in tableC.3 (Private use).

stringprep.**in_table_c4**(*code*)
Determine whether *code* is in tableC.4 (Non-character code points).

stringprep.**in_table_c5**(*code*)
Determine whether *code* is in tableC.5 (Surrogate codes).

stringprep.**in_table_c6**(*code*)
Determine whether *code* is in tableC.6 (Inappropriate for plain text).

stringprep.**in_table_c7**(*code*)
Determine whether *code* is in tableC.7 (Inappropriate for canonical representation).

stringprep.**in_table_c8**(*code*)
Determine whether *code* is in tableC.8 (Change display properties or are deprecated).

stringprep.**in_table_c9**(*code*)
Determine whether *code* is in tableC.9 (Tagging characters).

stringprep.**in_table_d1**(*code*)
Determine whether *code* is in tableD.1 (Characters with bidirectional property “R” or “AL”).

stringprep.**in_table_d2**(*code*)
Determine whether *code* is in tableD.2 (Characters with bidirectional property “L”).

6.7 readline — GNU readline interface

The *readline* module defines a number of functions to facilitate completion and reading/writing of history files from the Python interpreter. This module can be used directly, or via the *rlcompleter* module, which supports completion of Python identifiers at the interactive prompt. Settings made using this module affect the behaviour of both the interpreter's interactive prompt and the prompts offered by the built-in *input()* function.

Readline keybindings may be configured via an initialization file, typically `.inputrc` in your home directory. See Readline Init File in the GNU Readline manual for information about the format and allowable constructs of that file, and the capabilities of the Readline library in general.

Nota: The underlying Readline library API may be implemented by the `libedit` library instead of GNU readline. On macOS the *readline* module detects which library is being used at run time.

The configuration file for `libedit` is different from that of GNU readline. If you programmatically load configuration strings you can check for the text "libedit" in `readline.__doc__` to differentiate between GNU readline and libedit.

If you use *editline*/`libedit` readline emulation on macOS, the initialization file located in your home directory is named `.editrc`. For example, the following content in `~/.editrc` will turn ON *vi* keybindings and TAB completion:

```
python:bind -v
python:bind ^I rl_complete
```

6.7.1 Arquivo init

The following functions relate to the init file and user configuration:

`readline.`**`parse_and_bind`**`(`*string*`)`
Execute the init line provided in the *string* argument. This calls `rl_parse_and_bind()` in the underlying library.

`readline.`**`read_init_file`**`(`[*filename*]`)`
Execute a readline initialization file. The default filename is the last filename used. This calls `rl_read_init_file()` in the underlying library.

6.7.2 Line buffer

The following functions operate on the line buffer:

`readline.`**`get_line_buffer`**`()`
Return the current contents of the line buffer (`rl_line_buffer` in the underlying library).

`readline.`**`insert_text`**`(`*string*`)`
Insert text into the line buffer at the cursor position. This calls `rl_insert_text()` in the underlying library, but ignores the return value.

`readline.`**`redisplay`**`()`
Change what's displayed on the screen to reflect the current contents of the line buffer. This calls `rl_redisplay()` in the underlying library.

6.7.3 Arquivo de histórico

The following functions operate on a history file:

readline.**read_history_file**([*filename*])
Load a readline history file, and append it to the history list. The default filename is `~/.history`. This calls `read_history()` in the underlying library.

readline.**write_history_file**([*filename*])
Save the history list to a readline history file, overwriting any existing file. The default filename is `~/.history`. This calls `write_history()` in the underlying library.

readline.**append_history_file**(*nelements*[, *filename*])
Append the last *nelements* items of history to a file. The default filename is `~/.history`. The file must already exist. This calls `append_history()` in the underlying library. This function only exists if Python was compiled for a version of the library that supports it.

Novo na versão 3.5.

readline.**get_history_length**()
readline.**set_history_length**(*length*)
Set or return the desired number of lines to save in the history file. The *write_history_file()* function uses this value to truncate the history file, by calling `history_truncate_file()` in the underlying library. Negative values imply unlimited history file size.

6.7.4 History list

The following functions operate on a global history list:

readline.**clear_history**()
Clear the current history. This calls `clear_history()` in the underlying library. The Python function only exists if Python was compiled for a version of the library that supports it.

readline.**get_current_history_length**()
Return the number of items currently in the history. (This is different from *get_history_length()*, which returns the maximum number of lines that will be written to a history file.)

readline.**get_history_item**(*index*)
Return the current contents of history item at *index*. The item index is one-based. This calls `history_get()` in the underlying library.

readline.**remove_history_item**(*pos*)
Remove history item specified by its position from the history. The position is zero-based. This calls `remove_history()` in the underlying library.

readline.**replace_history_item**(*pos*, *line*)
Replace history item specified by its position with *line*. The position is zero-based. This calls `replace_history_entry()` in the underlying library.

readline.**add_history**(*line*)
Append *line* to the history buffer, as if it was the last line typed. This calls `add_history()` in the underlying library.

readline.**set_auto_history**(*enabled*)
Enable or disable automatic calls to `add_history()` when reading input via readline. The *enabled* argument should be a Boolean value that when true, enables auto history, and that when false, disables auto history.

Novo na versão 3.6.

CPython implementation detail: Auto history is enabled by default, and changes to this do not persist across multiple sessions.

6.7.5 Startup hooks

readline.**set_startup_hook**([*function*])
Set or remove the function invoked by the `rl_startup_hook` callback of the underlying library. If *function* is specified, it will be used as the new hook function; if omitted or `None`, any function already installed is removed. The hook is called with no arguments just before readline prints the first prompt.

readline.**set_pre_input_hook**([*function*])
Set or remove the function invoked by the `rl_pre_input_hook` callback of the underlying library. If *function* is specified, it will be used as the new hook function; if omitted or `None`, any function already installed is removed. The hook is called with no arguments after the first prompt has been printed and just before readline starts reading input characters. This function only exists if Python was compiled for a version of the library that supports it.

6.7.6 Completion

The following functions relate to implementing a custom word completion function. This is typically operated by the Tab key, and can suggest and automatically complete a word being typed. By default, Readline is set up to be used by *rlcompleter* to complete Python identifiers for the interactive interpreter. If the *readline* module is to be used with a custom completer, a different set of word delimiters should be set.

readline.**set_completer**([*function*])
Set or remove the completer function. If *function* is specified, it will be used as the new completer function; if omitted or `None`, any completer function already installed is removed. The completer function is called as `function(text, state)`, for *state* in `0`, `1`, `2`, ..., until it returns a non-string value. It should return the next possible completion starting with *text*.

The installed completer function is invoked by the *entry_func* callback passed to `rl_completion_matches()` in the underlying library. The *text* string comes from the first parameter to the `rl_attempted_completion_function` callback of the underlying library.

readline.**get_completer**()
Get the completer function, or `None` if no completer function has been set.

readline.**get_completion_type**()
Get the type of completion being attempted. This returns the `rl_completion_type` variable in the underlying library as an integer.

readline.**get_begidx**()
readline.**get_endidx**()
Get the beginning or ending index of the completion scope. These indexes are the *start* and *end* arguments passed to the `rl_attempted_completion_function` callback of the underlying library.

readline.**set_completer_delims**(*string*)
readline.**get_completer_delims**()
Set or get the word delimiters for completion. These determine the start of the word to be considered for completion (the completion scope). These functions access the `rl_completer_word_break_characters` variable in the underlying library.

readline.**set_completion_display_matches_hook**([*function*])
Set or remove the completion display function. If *function* is specified, it will be used as the new completion display function; if omitted or `None`, any completion display function already installed is removed. This sets or clears the `rl_completion_display_matches_hook` callback in the underlying library. The completion

display function is called as `function(substitution, [matches], longest_match_length)` once each time matches need to be displayed.

6.7.7 Exemplo

The following example demonstrates how to use the *readline* module's history reading and writing functions to automatically load and save a history file named `.python_history` from the user's home directory. The code below would normally be executed automatically during interactive sessions from the user's `PYTHONSTARTUP` file.

```
import atexit
import os
import readline

histfile = os.path.join(os.path.expanduser("~"), ".python_history")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow unruly
    readline.set_history_length(1000)
except FileNotFoundError:
    pass

atexit.register(readline.write_history_file, histfile)
```

This code is actually automatically run when Python is run in interactive mode (see *Configuração Readline*).

The following example achieves the same goal but supports concurrent interactive sessions, by only appending the new history.

```
import atexit
import os
import readline
histfile = os.path.join(os.path.expanduser("~"), ".python_history")

try:
    readline.read_history_file(histfile)
    h_len = readline.get_current_history_length()
except FileNotFoundError:
    open(histfile, 'wb').close()
    h_len = 0

def save(prev_h_len, histfile):
    new_h_len = readline.get_current_history_length()
    readline.set_history_length(1000)
    readline.append_history_file(new_h_len - prev_h_len, histfile)
atexit.register(save, h_len, histfile)
```

The following example extends the *code.InteractiveConsole* class to support history save/restore.

```
import atexit
import code
import os
import readline

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self, locals, filename)
```

(continua na próxima página)

(continuação da página anterior)

```
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except FileNotFoundError:
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.set_history_length(1000)
        readline.write_history_file(histfile)
```

6.8 `rlcompleter` — Função de completamento para GNU readline

Código Fonte: Lib/rlcompleter.py

O módulo *rlcompleter* define uma função de completamento adequada para o módulo *readline* completando identificadores Python válidos e palavras reservadas.

Quando este módulo é importado em uma plataforma Unix com o módulo *readline* disponível, uma instância da classe `Completer` é criada automaticamente e seu método `complete()` é definido como o completador de *readline*.

Exemplo:

```
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
readline.__doc__          readline.get_line_buffer(  readline.read_init_file(
readline.__file__         readline.insert_text(      readline.set_completer(
readline.__name__         readline.parse_and_bind(
>>> readline.
```

O módulo *rlcompleter* é projetado para uso com o modo interativo do Python. A menos que Python seja executado com a opção `-S`, o módulo é automaticamente importado e configurado (veja *Configuração Readline*).

Em plataformas sem *readline*, a classe `Completer` definida por este módulo ainda pode ser usada para propósitos personalizados.

6.8.1 Objetos Completer

Os objetos Completer têm o seguinte método:

`Completer.`**`complete`**(*text*, *state*)
Retorna o *state*-ésim completamento para *text*.

Se chamado para *text* que não inclui um caractere de ponto (`'.'`), ele será completado a partir dos nomes atualmente definidos em *__main__*, *builtins* e palavras reservadas (conforme definido pelo módulo *keyword*).

Se chamado por um nome pontilhado, vai tentar avaliar qualquer coisa sem efeitos colaterais óbvios (as funções não serão avaliadas, mas pode levantar chamadas para `__getattr__()`) até a última parte e encontrar correspondências para o resto por meio da função *dir()*. Qualquer exceção levantada durante a avaliação da expressão é capturada, silenciada e *None* é retornado.

CAPÍTULO 7

Serviços de Dados Binários

Os módulos descritos neste capítulo fornecem algumas operações de serviços básicos para manipulação de dados binários. Outras operações sobre dados binários, especificamente em relação a formatos de arquivo e protocolos de rede, são descritas nas seções relevantes.

Algumas bibliotecas descritas em *Serviços de Processamento de Texto* também funcionam com formatos binários compatíveis com ASCII (por exemplo `re`) ou com todos os dados binários (por exemplo `difflib`).

Além disso, consulte a documentação dos tipos de dados binários internos do Python em *Tipos de Sequência Binária — bytes, bytearray, memoryview*.

7.1 `struct` — Interpret bytes as packed binary data

Código Fonte: Lib/struct.py

This module performs conversions between Python values and C structs represented as Python `bytes` objects. This can be used in handling binary data stored in files or from network connections, among other sources. It uses *Format Strings* as compact descriptions of the layout of the C structs and the intended conversion to/from Python values.

Nota: By default, the result of packing a given C struct includes pad bytes in order to maintain proper alignment for the C types involved; similarly, alignment is taken into account when unpacking. This behavior is chosen so that the bytes of a packed struct correspond exactly to the layout in memory of the corresponding C struct. To handle platform-independent data formats or omit implicit pad bytes, use `standard` size and alignment instead of `native` size and alignment: see *Byte Order, Size, and Alignment* for details.

Several `struct` functions (and methods of `Struct`) take a *buffer* argument. This refers to objects that implement the bufferobjects and provide either a readable or read-writable buffer. The most common types used for that purpose are `bytes` and `bytearray`, but many other types that can be viewed as an array of bytes implement the buffer protocol, so that they can be read/filled without additional copying from a `bytes` object.

7.1.1 Funções e Exceções

The module defines the following exception and functions:

exception struct.**error**
Exception raised on various occasions; argument is a string describing what is wrong.

struct.**pack**(*format*, *v1*, *v2*, ...)
Return a bytes object containing the values *v1*, *v2*, … packed according to the format string *format*. The arguments must match the values required by the format exactly.

struct.**pack_into**(*format*, *buffer*, *offset*, *v1*, *v2*, ...)
Pack the values *v1*, *v2*, … according to the format string *format* and write the packed bytes into the writable buffer *buffer* starting at position *offset*. Note that *offset* is a required argument.

struct.**unpack**(*format*, *buffer*)
Unpack from the buffer *buffer* (presumably packed by `pack(format, ...)`) according to the format string *format*. The result is a tuple even if it contains exactly one item. The buffer's size in bytes must match the size required by the format, as reflected by `calcsize()`.

struct.**unpack_from**(*format*, *buffer*, *offset=0*)
Unpack from *buffer* starting at position *offset*, according to the format string *format*. The result is a tuple even if it contains exactly one item. The buffer's size in bytes, minus *offset*, must be at least the size required by the format, as reflected by `calcsize()`.

struct.**iter_unpack**(*format*, *buffer*)
Iteratively unpack from the buffer *buffer* according to the format string *format*. This function returns an iterator which will read equally-sized chunks from the buffer until all its contents have been consumed. The buffer's size in bytes must be a multiple of the size required by the format, as reflected by `calcsize()`.

Each iteration yields a tuple as specified by the format string.

Novo na versão 3.4.

struct.**calcsize**(*format*)
Return the size of the struct (and hence of the bytes object produced by `pack(format, ...)`) corresponding to the format string *format*.

7.1.2 Format Strings

Format strings are the mechanism used to specify the expected layout when packing and unpacking data. They are built up from *Caracteres Formatados*, which specify the type of data being packed/unpacked. In addition, there are special characters for controlling the *Byte Order, Size, and Alignment*.

Byte Order, Size, and Alignment

By default, C types are represented in the machine's native format and byte order, and properly aligned by skipping pad bytes if necessary (according to the rules used by the C compiler).

Alternatively, the first character of the format string can be used to indicate the byte order, size and alignment of the packed data, according to the following table:

Character	Ordem de bytes	Tamanho	Alinhamento
`@`	nativo	nativo	nativo
=	nativo	padrão	nenhum
<	little-endian	padrão	nenhum
>	big-endian	padrão	nenhum
!	rede (= big-endian)	padrão	nenhum

Se o primeiro caractere não for um destes, `'@'` é presumido.

Native byte order is big-endian or little-endian, depending on the host system. For example, Intel x86 and AMD64 (x86-64) are little-endian; Motorola 68000 and PowerPC G5 are big-endian; ARM and Intel Itanium feature switchable endianness (bi-endian). Use `sys.byteorder` to check the endianness of your system.

Native size and alignment are determined using the C compiler's `sizeof` expression. This is always combined with native byte order.

Standard size depends only on the format character; see the table in the *Caracteres Formatados* section.

Note the difference between `'@'` and `'='`: both use native byte order, but the size and alignment of the latter is standardized.

The form `'!'` is available for those poor souls who claim they can't remember whether network byte order is big-endian or little-endian.

There is no way to indicate non-native byte order (force byte-swapping); use the appropriate choice of `'<'` or `'>'`.

Notas:

(1) Padding is only automatically added between successive structure members. No padding is added at the beginning or the end of the encoded struct.

(2) No padding is added when using non-native size and alignment, e.g. with '<', '>', '=', and '!'.

(3) To align the end of a structure to the alignment requirement of a particular type, end the format with the code for that type with a repeat count of zero. See *Exemplos*.

Caracteres Formatados

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types. The 'Standard size' column refers to the size of the packed value in bytes when using standard size; that is, when the format string starts with one of `'<'`, `'>'`, `'!'` or `'='`. When using native size, the size of the packed value is platform-dependent.

Formatação	Tipo em C	Python type	Standard size	Notas
x	pad byte	no value		
c	char	bytes of length 1	1	
b	signed char	inteiro	1	(1), (2)
B	unsigned char	inteiro	1	(2)
?	_Bool	bool	1	(1)
h	short	inteiro	2	(2)
H	unsigned short	inteiro	2	(2)
i	int	inteiro	4	(2)
I	unsigned int	inteiro	4	(2)
l	long	inteiro	4	(2)
L	unsigned long	inteiro	4	(2)
q	long long	inteiro	8	(2)
Q	unsigned long long	inteiro	8	(2)
n	ssize_t	inteiro		(3)
N	size_t	inteiro		(3)
e	(6)	float	2	(4)
f	float	float	4	(4)
d	double	float	8	(4)
s	char[]	bytes		
p	char[]	bytes		
P	void *	inteiro		(5)

Alterado na versão 3.3: Added support for the `'n'` and `'N'` formats.

Alterado na versão 3.6: Added support for the `'e'` format.

Notas:

(1) The `'?'` conversion code corresponds to the `_Bool` type defined by C99. If this type is not available, it is simulated using a `char`. In standard mode, it is always represented by one byte.

(2) When attempting to pack a non-integer using any of the integer conversion codes, if the non-integer has a `__index__()` method then that method is called to convert the argument to an integer before packing.

Alterado na versão 3.2: Use of the `__index__()` method for non-integers is new in 3.2.

(3) The `'n'` and `'N'` conversion codes are only available for the native size (selected as the default or with the `'@'` byte order character). For the standard size, you can use whichever of the other integer formats fits your application.

(4) For the `'f'`, `'d'` and `'e'` conversion codes, the packed representation uses the IEEE 754 binary32, binary64 or binary16 format (for `'f'`, `'d'` or `'e'` respectively), regardless of the floating-point format used by the platform.

(5) The `'P'` format character is only available for the native byte ordering (selected as the default or with the `'@'` byte order character). The byte order character `'='` chooses to use little- or big-endian ordering based on the host system. The struct module does not interpret this as native ordering, so the `'P'` format is not available.

(6) The IEEE 754 binary16 "half precision" type was introduced in the 2008 revision of the IEEE 754 standard. It has a sign bit, a 5-bit exponent and 11-bit precision (with 10 bits explicitly stored), and can represent numbers between approximately `6.1e-05` and `6.5e+04` at full precision. This type is not widely supported by C compilers: on a typical machine, an unsigned short can be used for storage, but not for math operations. See the Wikipedia page on the half-precision floating-point format for more information.

A format character may be preceded by an integral repeat count. For example, the format string `'4h'` means exactly the same as `'hhhh'`.

Whitespace characters between formats are ignored; a count and its format must not contain whitespace though.

For the `'s'` format character, the count is interpreted as the length of the bytes, not a repeat count like for the other format characters; for example, `'10s'` means a single 10-byte string, while `'10c'` means 10 characters. If a count is not given, it defaults to 1. For packing, the string is truncated or padded with null bytes as appropriate to make it fit. For unpacking, the resulting bytes object always has exactly the specified number of bytes. As a special case, `'0s'` means a single, empty string (while `'0c'` means 0 characters).

When packing a value `x` using one of the integer formats (`'b'`, `'B'`, `'h'`, `'H'`, `'i'`, `'I'`, `'l'`, `'L'`, `'q'`, `'Q'`), if `x` is outside the valid range for that format then *`struct.error`* is raised.

Alterado na versão 3.1: In 3.0, some of the integer formats wrapped out-of-range values and raised *`DeprecationWarning`* instead of *`struct.error`*.

The `'p'` format character encodes a "Pascal string", meaning a short variable-length string stored in a *fixed number of bytes*, given by the count. The first byte stored is the length of the string, or 255, whichever is smaller. The bytes of the string follow. If the string passed in to *`pack()`* is too long (longer than the count minus 1), only the leading `count-1` bytes of the string are stored. If the string is shorter than `count-1`, it is padded with null bytes so that exactly count bytes in all are used. Note that for *`unpack()`*, the `'p'` format character consumes `count` bytes, but that the string returned can never contain more than 255 bytes.

For the `'?'` format character, the return value is either *`True`* or *`False`*. When packing, the truth value of the argument object is used. Either 0 or 1 in the native or standard bool representation will be packed, and any non-zero value will be `True` when unpacking.

Exemplos

Nota: All examples assume a native byte order, size, and alignment with a big-endian machine.

A basic example of packing/unpacking three integers:

```
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
b'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('hhl', b'\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('hhl')
8
```

Unpacked fields can be named by assigning them to variables or by wrapping the result in a named tuple:

```
>>> record = b'raymond   \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)

>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name=b'raymond   ', serialnum=4658, school=264, gradelevel=8)
```

The ordering of format characters may have an impact on size since the padding needed to satisfy alignment requirements is different:

```
>>> pack('ci', b'*', 0x12131415)
b'*\x00\x00\x00\x12\x13\x14\x15'
>>> pack('ic', 0x12131415, b'*')
b'\x12\x13\x14\x15*'
>>> calcsize('ci')
```

(continua na próxima página)

(continuação da página anterior)

```
8
>>> calcsize('ic')
5
```

The following format `'llh0l'` specifies two pad bytes at the end, assuming longs are aligned on 4-byte boundaries:

```
>>> pack('llh0l', 1, 2, 3)
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

This only works when native size and alignment are in effect; standard size and alignment does not enforce any alignment.

Ver também:

Module ***array*** Packed binary storage of homogeneous data.

Módulo ***xdrlib*** Packing and unpacking of XDR data.

7.1.3 Classes

The *struct* module also defines the following type:

class struct.**Struct**(*format*)

Return a new Struct object which writes and reads binary data according to the format string *format*. Creating a Struct object once and calling its methods is more efficient than calling the *struct* functions with the same format since the format string only needs to be compiled once.

Nota: The compiled versions of the most recent format strings passed to *Struct* and the module-level functions are cached, so programs that use only a few format strings needn't worry about reusing a single *Struct* instance.

Compiled Struct objects support the following methods and attributes:

pack(*v1*, *v2*, ...)

Identical to the *pack()* function, using the compiled format. (`len(result)` will equal *size*.)

pack_into(*buffer*, *offset*, *v1*, *v2*, ...)

Identical to the *pack_into()* function, using the compiled format.

unpack(*buffer*)

Identical to the *unpack()* function, using the compiled format. The buffer's size in bytes must equal *size*.

unpack_from(*buffer*, *offset=0*)

Identical to the *unpack_from()* function, using the compiled format. The buffer's size in bytes, minus *offset*, must be at least *size*.

iter_unpack(*buffer*)

Identical to the *iter_unpack()* function, using the compiled format. The buffer's size in bytes must be a multiple of *size*.

Novo na versão 3.4.

format

The format string used to construct this Struct object.

Alterado na versão 3.7: The format string type is now *str* instead of *bytes*.

size

The calculated size of the struct (and hence of the bytes object produced by the *pack()* method) corresponding to *format*.

7.2 codecs — Codec registry and base classes

Código Fonte: Lib/codecs.py

This module defines base classes for standard Python codecs (encoders and decoders) and provides access to the internal Python codec registry, which manages the codec and error handling lookup process. Most standard codecs are *text encodings*, which encode text to bytes, but there are also codecs provided that encode text to text, and bytes to bytes. Custom codecs may encode and decode between arbitrary types, but some module features are restricted to use specifically with *text encodings*, or with codecs that encode to `bytes`.

The module defines the following functions for encoding and decoding with any codec:

codecs.**encode**(*obj*, *encoding='utf-8'*, *errors='strict'*)
Encodes *obj* using the codec registered for *encoding*.

Errors may be given to set the desired error handling scheme. The default error handler is `'strict'` meaning that encoding errors raise `ValueError` (or a more codec specific subclass, such as `UnicodeEncodeError`). Refer to *Codec Base Classes* for more information on codec error handling.

codecs.**decode**(*obj*, *encoding='utf-8'*, *errors='strict'*)
Decodes *obj* using the codec registered for *encoding*.

Errors may be given to set the desired error handling scheme. The default error handler is `'strict'` meaning that decoding errors raise `ValueError` (or a more codec specific subclass, such as `UnicodeDecodeError`). Refer to *Codec Base Classes* for more information on codec error handling.

The full details for each codec can also be looked up directly:

codecs.**lookup**(*encoding*)
Looks up the codec info in the Python codec registry and returns a `CodecInfo` object as defined below.

Encodings are first looked up in the registry's cache. If not found, the list of registered search functions is scanned. If no `CodecInfo` object is found, a `LookupError` is raised. Otherwise, the `CodecInfo` object is stored in the cache and returned to the caller.

class codecs.**CodecInfo**(*encode*, *decode*, *streamreader=None*, *streamwriter=None*, *incrementalencoder=None*, *incrementaldecoder=None*, *name=None*)
Codec details when looking up the codec registry. The constructor arguments are stored in attributes of the same name:

name
The name of the encoding.

encode
decode
The stateless encoding and decoding functions. These must be functions or methods which have the same interface as the `encode()` and `decode()` methods of Codec instances (see *Codec Interface*). The functions or methods are expected to work in a stateless mode.

incrementalencoder
incrementaldecoder
Incremental encoder and decoder classes or factory functions. These have to provide the interface defined by the base classes `IncrementalEncoder` and `IncrementalDecoder`, respectively. Incremental codecs can maintain state.

streamwriter
streamreader
Stream writer and reader classes or factory functions. These have to provide the interface defined by the base classes `StreamWriter` and `StreamReader`, respectively. Stream codecs can maintain state.

To simplify access to the various codec components, the module provides these additional functions which use *lookup()* for the codec lookup:

codecs.**getencoder**(*encoding*)
Look up the codec for the given encoding and return its encoder function.

Raises a *LookupError* in case the encoding cannot be found.

codecs.**getdecoder**(*encoding*)
Look up the codec for the given encoding and return its decoder function.

Raises a *LookupError* in case the encoding cannot be found.

codecs.**getincrementalencoder**(*encoding*)
Look up the codec for the given encoding and return its incremental encoder class or factory function.

Raises a *LookupError* in case the encoding cannot be found or the codec doesn't support an incremental encoder.

codecs.**getincrementaldecoder**(*encoding*)
Look up the codec for the given encoding and return its incremental decoder class or factory function.

Raises a *LookupError* in case the encoding cannot be found or the codec doesn't support an incremental decoder.

codecs.**getreader**(*encoding*)
Look up the codec for the given encoding and return its *StreamReader* class or factory function.

Raises a *LookupError* in case the encoding cannot be found.

codecs.**getwriter**(*encoding*)
Look up the codec for the given encoding and return its *StreamWriter* class or factory function.

Raises a *LookupError* in case the encoding cannot be found.

Custom codecs are made available by registering a suitable codec search function:

codecs.**register**(*search_function*)
Register a codec search function. Search functions are expected to take one argument, being the encoding name in all lower case letters, and return a *CodecInfo* object. In case a search function cannot find a given encoding, it should return `None`.

Nota: Search function registration is not currently reversible, which may cause problems in some cases, such as unit testing or module reloading.

While the builtin *open()* and the associated *io* module are the recommended approach for working with encoded text files, this module provides additional utility functions and classes that allow the use of a wider range of codecs when working with binary files:

codecs.**open**(*filename*, *mode='r'*, *encoding=None*, *errors='strict'*, *buffering=1*)
Open an encoded file using the given *mode* and return an instance of *StreamReaderWriter*, providing transparent encoding/decoding. The default file mode is `'r'`, meaning to open the file in read mode.

Nota: Underlying encoded files are always opened in binary mode. No automatic conversion of `'\n'` is done on reading and writing. The *mode* argument may be any binary mode acceptable to the built-in *open()* function; the `'b'` is automatically added.

---

encoding specifies the encoding which is to be used for the file. Any encoding that encodes to and decodes from bytes is allowed, and the data types supported by the file methods depend on the codec used.

errors may be given to define the error handling. It defaults to `'strict'` which causes a *ValueError* to be raised in case an encoding error occurs.

buffering has the same meaning as for the built-in `open()` function. It defaults to line buffered.

codecs.**EncodedFile**(*file*, *data_encoding*, *file_encoding=None*, *errors='strict'*)
Return a `StreamRecoder` instance, a wrapped version of *file* which provides transparent transcoding. The original file is closed when the wrapped version is closed.

Data written to the wrapped file is decoded according to the given *data_encoding* and then written to the original file as bytes using *file_encoding*. Bytes read from the original file are decoded according to *file_encoding*, and the result is encoded using *data_encoding*.

If *file_encoding* is not given, it defaults to *data_encoding*.

errors may be given to define the error handling. It defaults to `'strict'`, which causes `ValueError` to be raised in case an encoding error occurs.

codecs.**iterencode**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)
Uses an incremental encoder to iteratively encode the input provided by *iterator*. This function is a *generator*. The *errors* argument (as well as any other keyword argument) is passed through to the incremental encoder.

This function requires that the codec accept text `str` objects to encode. Therefore it does not support bytes-to-bytes encoders such as `base64_codec`.

codecs.**iterdecode**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)
Uses an incremental decoder to iteratively decode the input provided by *iterator*. This function is a *generator*. The *errors* argument (as well as any other keyword argument) is passed through to the incremental decoder.

This function requires that the codec accept `bytes` objects to decode. Therefore it does not support text-to-text encoders such as `rot_13`, although `rot_13` may be used equivalently with `iterencode()`.

The module also provides the following constants which are useful for reading and writing to platform dependent files:

codecs.**BOM**
codecs.**BOM_BE**
codecs.**BOM_LE**
codecs.**BOM_UTF8**
codecs.**BOM_UTF16**
codecs.**BOM_UTF16_BE**
codecs.**BOM_UTF16_LE**
codecs.**BOM_UTF32**
codecs.**BOM_UTF32_BE**
codecs.**BOM_UTF32_LE**
These constants define various byte sequences, being Unicode byte order marks (BOMs) for several encodings. They are used in UTF-16 and UTF-32 data streams to indicate the byte order used, and in UTF-8 as a Unicode signature. `BOM_UTF16` is either `BOM_UTF16_BE` or `BOM_UTF16_LE` depending on the platform's native byte order, `BOM` is an alias for `BOM_UTF16`, `BOM_LE` for `BOM_UTF16_LE` and `BOM_BE` for `BOM_UTF16_BE`. The others represent the BOM in UTF-8 and UTF-32 encodings.

7.2.1 Codec Base Classes

The `codecs` module defines a set of base classes which define the interfaces for working with codec objects, and can also be used as the basis for custom codec implementations.

Each codec has to define four interfaces to make it usable as codec in Python: stateless encoder, stateless decoder, stream reader and stream writer. The stream reader and writers typically reuse the stateless encoder/decoder to implement the file protocols. Codec authors also need to define how the codec will handle encoding and decoding errors.

Error Handlers

To simplify and standardize error handling, codecs may implement different error handling schemes by accepting the *errors* string argument. The following string values are defined and implemented by all standard Python codecs:

Valor	Significado
`'strict'`	Raise *UnicodeError* (or a subclass); this is the default. Implemented in *strict_errors()*.
`'ignore'`	Ignore the malformed data and continue without further notice. Implemented in *ignore_errors()*.

The following error handlers are only applicable to *text encodings*:

Valor	Significado
`'replace'`	Replace with a suitable replacement marker; Python will use the official `U+FFFD` REPLACEMENT CHARACTER for the built-in codecs on decoding, and '?' on encoding. Implemented in *replace_errors()*.
`'xmlcharrefreplace'`	Replace with the appropriate XML character reference (only for encoding). Implemented in *xmlcharrefreplace_errors()*.
`'backslashreplace'`	Replace with backslashed escape sequences. Implemented in *backslashreplace_errors()*.
`'namereplace'`	Replace with `\N{...}` escape sequences (only for encoding). Implemented in *namereplace_errors()*.
`'surrogateescape'`	On decoding, replace byte with individual surrogate code ranging from `U+DC80` to `U+DCFF`. This code will then be turned back into the same byte when the `'surrogateescape'` error handler is used when encoding the data. (See **PEP 383** for more.)

In addition, the following error handler is specific to the given codecs:

Valor	Codecs	Significado
`'surrogatepass'`	utf-8, utf-16, utf-32, utf-16-be, utf-16-le, utf-32-be, utf-32-le	Allow encoding and decoding of surrogate codes. These codecs normally treat the presence of surrogates as an error.

Novo na versão 3.1: The `'surrogateescape'` and `'surrogatepass'` error handlers.

Alterado na versão 3.4: The `'surrogatepass'` error handlers now works with utf-16* and utf-32* codecs.

Novo na versão 3.5: The `'namereplace'` error handler.

Alterado na versão 3.5: The `'backslashreplace'` error handlers now works with decoding and translating.

The set of allowed values can be extended by registering a new named error handler:

`codecs.`**`register_error`**(*name*, *error_handler*)

Register the error handling function *error_handler* under the name *name*. The *error_handler* argument will be called during encoding and decoding in case of an error, when *name* is specified as the errors parameter.

For encoding, *error_handler* will be called with a *UnicodeEncodeError* instance, which contains information about the location of the error. The error handler must either raise this or a different exception, or return a tuple with a replacement for the unencodable part of the input and a position where encoding should continue. The replacement may be either *str* or *bytes*. If the replacement is bytes, the encoder will simply copy them into the output buffer. If the replacement is a string, the encoder will encode the replacement. Encoding continues on original input at the specified position. Negative position values will be treated as being relative to the end of the input string. If the resulting position is out of bound an *IndexError* will be raised.

Decoding and translating works similarly, except *UnicodeDecodeError* or *UnicodeTranslateError* will be passed to the handler and that the replacement from the error handler will be put into the output directly.

Previously registered error handlers (including the standard error handlers) can be looked up by name:

codecs.**lookup_error**(*name*)
Return the error handler previously registered under the name *name*.

Raises a *LookupError* in case the handler cannot be found.

The following standard error handlers are also made available as module level functions:

codecs.**strict_errors**(*exception*)
Implements the `'strict'` error handling: each encoding or decoding error raises a *UnicodeError*.

codecs.**replace_errors**(*exception*)
Implements the `'replace'` error handling (for *text encodings* only): substitutes `'?'` for encoding errors (to be encoded by the codec), and `'\ufffd'` (the Unicode replacement character) for decoding errors.

codecs.**ignore_errors**(*exception*)
Implements the `'ignore'` error handling: malformed data is ignored and encoding or decoding is continued without further notice.

codecs.**xmlcharrefreplace_errors**(*exception*)
Implements the `'xmlcharrefreplace'` error handling (for encoding with *text encodings* only): the unencodable character is replaced by an appropriate XML character reference.

codecs.**backslashreplace_errors**(*exception*)
Implements the `'backslashreplace'` error handling (for *text encodings* only): malformed data is replaced by a backslashed escape sequence.

codecs.**namereplace_errors**(*exception*)
Implements the `'namereplace'` error handling (for encoding with *text encodings* only): the unencodable character is replaced by a `\N{...}` escape sequence.

Novo na versão 3.5.

Stateless Encoding and Decoding

The base `Codec` class defines these methods which also define the function interfaces of the stateless encoder and decoder:

Codec.**encode**(*input*[, *errors*])
Encodes the object *input* and returns a tuple (output object, length consumed). For instance, *text encoding* converts a string object to a bytes object using a particular character set encoding (e.g., `cp1252` or `iso-8859-1`).

The *errors* argument defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the `Codec` instance. Use *StreamWriter* for codecs which have to keep state in order to make encoding efficient.

The encoder must be able to handle zero length input and return an empty object of the output object type in this situation.

Codec.**decode**(*input*[, *errors*])
Decodes the object *input* and returns a tuple (output object, length consumed). For instance, for a *text encoding*, decoding converts a bytes object encoded using a particular character set encoding to a string object.

For text encodings and bytes-to-bytes codecs, *input* must be a bytes object or one which provides the read-only buffer interface – for example, buffer objects and memory mapped files.

The *errors* argument defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the `Codec` instance. Use *StreamReader* for codecs which have to keep state in order to make decoding efficient.

The decoder must be able to handle zero length input and return an empty object of the output object type in this situation.

Incremental Encoding and Decoding

The `IncrementalEncoder` and `IncrementalDecoder` classes provide the basic interface for incremental encoding and decoding. Encoding/decoding the input isn't done with one call to the stateless encoder/decoder function, but with multiple calls to the `encode()`/`decode()` method of the incremental encoder/decoder. The incremental encoder/decoder keeps track of the encoding/decoding process during method calls.

The joined output of calls to the `encode()`/`decode()` method is the same as if all the single inputs were joined into one, and this input was encoded/decoded with the stateless encoder/decoder.

IncrementalEncoder Objects

The `IncrementalEncoder` class is used for encoding an input in multiple steps. It defines the following methods which every incremental encoder must define in order to be compatible with the Python codec registry.

class codecs.**IncrementalEncoder**(*errors='strict'*)
Constructor for an `IncrementalEncoder` instance.

All incremental encoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The `IncrementalEncoder` may implement different error handling schemes by providing the *errors* keyword argument. See *Error Handlers* for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `IncrementalEncoder` object.

encode(*object*[, *final*])
Encodes *object* (taking the current state of the encoder into account) and returns the resulting encoded object. If this is the last call to `encode()` *final* must be true (the default is false).

reset()
Reset the encoder to the initial state. The output is discarded: call `.encode(object, final=True)`, passing an empty byte or text string if necessary, to reset the encoder and to get the output.

getstate()
Return the current state of the encoder which must be an integer. The implementation should make sure that `0` is the most common state. (States that are more complicated than integers can be converted into an integer by marshaling/pickling the state and encoding the bytes of the resulting string into an integer.)

setstate(*state*)
Set the state of the encoder to *state*. *state* must be an encoder state returned by `getstate()`.

IncrementalDecoder Objects

The `IncrementalDecoder` class is used for decoding an input in multiple steps. It defines the following methods which every incremental decoder must define in order to be compatible with the Python codec registry.

class codecs.**IncrementalDecoder**(*errors='strict'*)
Constructor for an `IncrementalDecoder` instance.

All incremental decoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *IncrementalDecoder* may implement different error handling schemes by providing the *errors* keyword argument. See *Error Handlers* for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the *IncrementalDecoder* object.

decode(*object*[, *final*])
Decodes *object* (taking the current state of the decoder into account) and returns the resulting decoded object. If this is the last call to *decode()* *final* must be true (the default is false). If *final* is true the decoder must decode the input completely and must flush all buffers. If this isn't possible (e.g. because of incomplete byte sequences at the end of the input) it must initiate error handling just like in the stateless case (which might raise an exception).

reset()
Reset the decoder to the initial state.

getstate()
Return the current state of the decoder. This must be a tuple with two items, the first must be the buffer containing the still undecoded input. The second must be an integer and can be additional state info. (The implementation should make sure that `0` is the most common additional state info.) If this additional state info is `0` it must be possible to set the decoder to the state which has no input buffered and `0` as the additional state info, so that feeding the previously buffered input to the decoder returns it to the previous state without producing any output. (Additional state info that is more complicated than integers can be converted into an integer by marshaling/pickling the info and encoding the bytes of the resulting string into an integer.)

setstate(*state*)
Set the state of the decoder to *state*. *state* must be a decoder state returned by *getstate()*.

Stream Encoding and Decoding

The *StreamWriter* and *StreamReader* classes provide generic working interfaces which can be used to implement new encoding submodules very easily. See `encodings.utf_8` for an example of how this is done.

StreamWriter Objects

The *StreamWriter* class is a subclass of `Codec` and defines the following methods which every stream writer must define in order to be compatible with the Python codec registry.

class codecs.**StreamWriter**(*stream*, *errors='strict'*)
Constructor for a *StreamWriter* instance.

All stream writers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *stream* argument must be a file-like object open for writing text or binary data, as appropriate for the specific codec.

The *StreamWriter* may implement different error handling schemes by providing the *errors* keyword argument. See *Error Handlers* for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the *StreamWriter* object.

write(*object*)
Writes the object's contents encoded to the stream.

writelines(*list*)
Writes the concatenated list of strings to the stream (possibly by reusing the `write()` method). The standard bytes-to-bytes codecs do not support this method.

reset()
Flushes and resets the codec buffers used for keeping state.

Chamar este método deve garantir que os dados na saída estejam num estado limpo, que permite anexar novos dados sem ter que verificar novamente todo o fluxo para recuperar o estado.

In addition to the above methods, the `StreamWriter` must also inherit all other methods and attributes from the underlying stream.

StreamReader Objects

The `StreamReader` class is a subclass of `Codec` and defines the following methods which every stream reader must define in order to be compatible with the Python codec registry.

class codecs.**StreamReader**(*stream*, *errors='strict'*)
Constructor for a `StreamReader` instance.

All stream readers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *stream* argument must be a file-like object open for reading text or binary data, as appropriate for the specific codec.

The `StreamReader` may implement different error handling schemes by providing the *errors* keyword argument. See *Error Handlers* for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `StreamReader` object.

The set of allowed values for the *errors* argument can be extended with `register_error()`.

read([*size*[, *chars*[, *firstline*]]])
Decodes data from the stream and returns the resulting object.

The *chars* argument indicates the number of decoded code points or bytes to return. The `read()` method will never return more data than requested, but it might return less, if there is not enough available.

The *size* argument indicates the approximate maximum number of encoded bytes or code points to read for decoding. The decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. This parameter is intended to prevent having to decode huge files in one step.

The *firstline* flag indicates that it would be sufficient to only return the first line, if there are decoding errors on later lines.

The method should use a greedy read strategy meaning that it should read as much data as is allowed within the definition of the encoding and the given size, e.g. if optional encoding endings or state markers are available on the stream, these should be read too.

readline([*size*[, *keepends*]])
Read one line from the input stream and return the decoded data.

size, if given, is passed as size argument to the stream's `read()` method.

If *keepends* is false line-endings will be stripped from the lines returned.

readlines([*sizehint*[, *keepends*]])
Read all lines available on the input stream and return them as a list of lines.

Line-endings are implemented using the codec's `decode()` method and are included in the list entries if *keepends* is true.

sizehint, if given, is passed as the *size* argument to the stream's `read()` method.

reset()
Resets the codec buffers used for keeping state.

Note that no stream repositioning should take place. This method is primarily intended to be able to recover from decoding errors.

In addition to the above methods, the `StreamReader` must also inherit all other methods and attributes from the underlying stream.

StreamReaderWriter Objects

The `StreamReaderWriter` is a convenience class that allows wrapping streams which work in both read and write modes.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

class codecs.**StreamReaderWriter**(*stream*, *Reader*, *Writer*, *errors='strict'*)
Creates a `StreamReaderWriter` instance. *stream* must be a file-like object. *Reader* and *Writer* must be factory functions or classes providing the `StreamReader` and `StreamWriter` interface resp. Error handling is done in the same way as defined for the stream readers and writers.

`StreamReaderWriter` instances define the combined interfaces of `StreamReader` and `StreamWriter` classes. They inherit all other methods and attributes from the underlying stream.

StreamRecoder Objects

The `StreamRecoder` translates data from one encoding to another, which is sometimes useful when dealing with different encoding environments.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

class codecs.**StreamRecoder**(*stream*, *encode*, *decode*, *Reader*, *Writer*, *errors='strict'*)
Creates a `StreamRecoder` instance which implements a two-way conversion: *encode* and *decode* work on the frontend — the data visible to code calling `read()` and `write()`, while *Reader* and *Writer* work on the backend — the data in *stream*.

You can use these objects to do transparent transcodings, e.g., from Latin-1 to UTF-8 and back.

The *stream* argument must be a file-like object.

The *encode* and *decode* arguments must adhere to the `Codec` interface. *Reader* and *Writer* must be factory functions or classes providing objects of the `StreamReader` and `StreamWriter` interface respectively.

Error handling is done in the same way as defined for the stream readers and writers.

`StreamRecoder` instances define the combined interfaces of `StreamReader` and `StreamWriter` classes. They inherit all other methods and attributes from the underlying stream.

7.2.2 Encodings and Unicode

Strings are stored internally as sequences of code points in range `0x0–0x10FFFF`. (See **PEP 393** for more details about the implementation.) Once a string object is used outside of CPU and memory, endianness and how these arrays are stored as bytes become an issue. As with other codecs, serialising a string into a sequence of bytes is known as *encoding*, and recreating the string from the sequence of bytes is known as *decoding*. There are a variety of different text serialisation codecs, which are collectivity referred to as *text encodings*.

The simplest text encoding (called `'latin-1'` or `'iso-8859-1'`) maps the code points 0–255 to the bytes `0x0–0xff`, which means that a string object that contains code points above `U+00FF` can't be encoded with this codec. Doing so will raise a *`UnicodeEncodeError`* that looks like the following (although the details of the error message may differ): `UnicodeEncodeError: 'latin-1' codec can't encode character '\u1234' in position 3: ordinal not in range(256).`

There's another group of encodings (the so called charmap encodings) that choose a different subset of all Unicode code points and how these code points are mapped to the bytes `0x0–0xff`. To see how this is done simply open e.g. `encodings/cp1252.py` (which is an encoding that is used primarily on Windows). There's a string constant with 256 characters that shows you which character is mapped to which byte value.

All of these encodings can only encode 256 of the 1114112 code points defined in Unicode. A simple and straightforward way that can store each Unicode code point, is to store each code point as four consecutive bytes. There are two possibilities: store the bytes in big endian or in little endian order. These two encodings are called `UTF-32-BE` and `UTF-32-LE` respectively. Their disadvantage is that if e.g. you use `UTF-32-BE` on a little endian machine you will always have to swap bytes on encoding and decoding. `UTF-32` avoids this problem: bytes will always be in natural endianness. When these bytes are read by a CPU with a different endianness, then bytes have to be swapped though. To be able to detect the endianness of a `UTF-16` or `UTF-32` byte sequence, there's the so called BOM ("Byte Order Mark"). This is the Unicode character `U+FEFF`. This character can be prepended to every `UTF-16` or `UTF-32` byte sequence. The byte swapped version of this character (`0xFFFE`) is an illegal character that may not appear in a Unicode text. So when the first character in an `UTF-16` or `UTF-32` byte sequence appears to be a `U+FFFE` the bytes have to be swapped on decoding. Unfortunately the character `U+FEFF` had a second purpose as a `ZERO WIDTH NO-BREAK SPACE`: a character that has no width and doesn't allow a word to be split. It can e.g. be used to give hints to a ligature algorithm. With Unicode 4.0 using `U+FEFF` as a `ZERO WIDTH NO-BREAK SPACE` has been deprecated (with `U+2060` (`WORD JOINER`) assuming this role). Nevertheless Unicode software still must be able to handle `U+FEFF` in both roles: as a BOM it's a device to determine the storage layout of the encoded bytes, and vanishes once the byte sequence has been decoded into a string; as a `ZERO WIDTH NO-BREAK SPACE` it's a normal character that will be decoded like any other.

There's another encoding that is able to encoding the full range of Unicode characters: UTF-8. UTF-8 is an 8-bit encoding, which means there are no issues with byte order in UTF-8. Each byte in a UTF-8 byte sequence consists of two parts: marker bits (the most significant bits) and payload bits. The marker bits are a sequence of zero to four `1` bits followed by a `0` bit. Unicode characters are encoded like this (with x being payload bits, which when concatenated give the Unicode character):

Range	Encoding
`U-00000000` ... `U-0000007F`	0xxxxxxx
`U-00000080` ... `U-000007FF`	110xxxxx 10xxxxxx
`U-00000800` ... `U-0000FFFF`	1110xxxx 10xxxxxx 10xxxxxx
`U-00010000` ... `U-0010FFFF`	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

The least significant bit of the Unicode character is the rightmost x bit.

As UTF-8 is an 8-bit encoding no BOM is required and any `U+FEFF` character in the decoded string (even if it's the first character) is treated as a `ZERO WIDTH NO-BREAK SPACE`.

Without external information it's impossible to reliably determine which encoding was used for encoding a string. Each charmap encoding can decode any random byte sequence. However that's not possible with UTF-8, as UTF-8 byte

sequences have a structure that doesn't allow arbitrary byte sequences. To increase the reliability with which a UTF-8 encoding can be detected, Microsoft invented a variant of UTF-8 (that Python 2.5 calls `"utf-8-sig"`) for its Notepad program: Before any of the Unicode characters is written to the file, a UTF-8 encoded BOM (which looks like this as a byte sequence: `0xef`, `0xbb`, `0xbf`) is written. As it's rather improbable that any charmap encoded file starts with these byte values (which would e.g. map to

> LATIN SMALL LETTER I WITH DIAERESIS
> RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
> INVERTED QUESTION MARK

in iso-8859-1), this increases the probability that a `utf-8-sig` encoding can be correctly guessed from the byte sequence. So here the BOM is not used to be able to determine the byte order used for generating the byte sequence, but as a signature that helps in guessing the encoding. On encoding the utf-8-sig codec will write `0xef`, `0xbb`, `0xbf` as the first three bytes to the file. On decoding `utf-8-sig` will skip those three bytes if they appear as the first three bytes in the file. In UTF-8, the use of the BOM is discouraged and should generally be avoided.

7.2.3 Standard Encodings

Python comes with a number of codecs built-in, either implemented as C functions or with dictionaries as mapping tables. The following table lists the codecs by name, together with a few common aliases, and the languages for which the encoding is likely used. Neither the list of aliases nor the list of languages is meant to be exhaustive. Notice that spelling alternatives that only differ in case or use a hyphen instead of an underscore are also valid aliases; therefore, e.g. `'utf-8'` is a valid alias for the `'utf_8'` codec.

CPython implementation detail: Some common encodings can bypass the codecs lookup machinery to improve performance. These optimization opportunities are only recognized by CPython for a limited set of (case insensitive) aliases: utf-8, utf8, latin-1, latin1, iso-8859-1, iso8859-1, mbcs (Windows only), ascii, us-ascii, utf-16, utf16, utf-32, utf32, and the same using underscores instead of dashes. Using alternative aliases for these encodings may result in slower execution.

Alterado na versão 3.6: Optimization opportunity recognized for us-ascii.

Many of the character sets support the same languages. They vary in individual characters (e.g. whether the EURO SIGN is supported or not), and in the assignment of characters to code positions. For the European languages in particular, the following variants typically exist:

- an ISO 8859 codeset
- a Microsoft Windows code page, which is typically derived from an 8859 codeset, but replaces control characters with additional graphic characters
- an IBM EBCDIC code page
- an IBM PC code page, which is ASCII compatible

Codec	Aliases	Languages
ascii	646, us-ascii	English
big5	big5-tw, csbig5	Traditional Chinese
big5hkscs	big5-hkscs, hkscs	Traditional Chinese
cp037	IBM037, IBM039	English
cp273	273, IBM273, csIBM273	German Novo na versão 3.4.
cp424	EBCDIC-CP-HE, IBM424	Hebrew
cp437	437, IBM437	English
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	Western Europe
cp720		Arabic

Continuação na próxima página

Tabela 1 – continuação da página anterior

Codec	Aliases	Languages
cp737		Greek
cp775	IBM775	Baltic languages
cp850	850, IBM850	Western Europe
cp852	852, IBM852	Central and Eastern Europe
cp855	855, IBM855	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
cp856		Hebrew
cp857	857, IBM857	Turkish
cp858	858, IBM858	Western Europe
cp860	860, IBM860	Portuguese
cp861	861, CP-IS, IBM861	Icelandic
cp862	862, IBM862	Hebrew
cp863	863, IBM863	Canadian
cp864	IBM864	Arabic
cp865	865, IBM865	Danish, Norwegian
cp866	866, IBM866	Russian
cp869	869, CP-GR, IBM869	Greek
cp874		Thai
cp875		Greek
cp932	932, ms932, mskanji, ms-kanji	Japanese
cp949	949, ms949, uhc	Korean
cp950	950, ms950	Traditional Chinese
cp1006		Urdu
cp1026	ibm1026	Turkish
cp1125	1125, ibm1125, cp866u, ruscii	Ukrainian Novo na versão 3.4.
cp1140	ibm1140	Western Europe
cp1250	windows-1250	Central and Eastern Europe
cp1251	windows-1251	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
cp1252	windows-1252	Western Europe
cp1253	windows-1253	Greek
cp1254	windows-1254	Turkish
cp1255	windows-1255	Hebrew
cp1256	windows-1256	Arabic
cp1257	windows-1257	Baltic languages
cp1258	windows-1258	Vietnamese
cp65001		Windows only: Windows UTF-8 (`CP_UTF8`) Novo na versão 3.3.
euc_jp	eucjp, ujis, u-jis	Japanese
euc_jis_2004	jisx0213, eucjis2004	Japanese
euc_jisx0213	eucjisx0213	Japanese
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	Korean
gb2312	chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58	Simplified Chinese
gbk	936, cp936, ms936	Unified Chinese

Continuação na próxima página

Tabela 1 – continuação da página anterior

Codec	Aliases	Languages
gb18030	gb18030-2000	Unified Chinese
hz	hzgb, hz-gb, hz-gb-2312	Simplified Chinese
iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	Japanese
iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	Japanese
iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	Japanese, Korean, Simplified Chinese, Western Europe, Greek
iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	Japanese
iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	Japanese
iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	Japanese
iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	Korean
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	Western Europe
iso8859_2	iso-8859-2, latin2, L2	Central and Eastern Europe
iso8859_3	iso-8859-3, latin3, L3	Esperanto, Maltese
iso8859_4	iso-8859-4, latin4, L4	Baltic languages
iso8859_5	iso-8859-5, cyrillic	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
iso8859_6	iso-8859-6, arabic	Arabic
iso8859_7	iso-8859-7, greek, greek8	Greek
iso8859_8	iso-8859-8, hebrew	Hebrew
iso8859_9	iso-8859-9, latin5, L5	Turkish
iso8859_10	iso-8859-10, latin6, L6	Nordic languages
iso8859_11	iso-8859-11, thai	Thai languages
iso8859_13	iso-8859-13, latin7, L7	Baltic languages
iso8859_14	iso-8859-14, latin8, L8	Celtic languages
iso8859_15	iso-8859-15, latin9, L9	Western Europe
iso8859_16	iso-8859-16, latin10, L10	South-Eastern Europe
johab	cp1361, ms1361	Korean
koi8_r		Russian
koi8_t		Tajik Novo na versão 3.5.
koi8_u		Ukrainian
kz1048	kz_1048, strk1048_2002, rk1048	Kazakh Novo na versão 3.5.
mac_cyrillic	maccyrillic	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
mac_greek	macgreek	Greek
mac_iceland	maciceland	Icelandic
mac_latin2	maclatin2, maccentraleurope	Central and Eastern Europe
mac_roman	macroman, macintosh	Western Europe
mac_turkish	macturkish	Turkish
ptcp154	csptcp154, pt154, cp154, cyrillic-asian	Kazakh
shift_jis	csshiftjis, shiftjis, sjis, s_jis	Japanese
shift_jis_2004	shiftjis2004, sjis_2004, sjis2004	Japanese
shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	Japanese
utf_32	U32, utf32	todas linguagens
utf_32_be	UTF-32BE	todas linguagens

Continuação na próxima página

Tabela 1 – continuação da página anterior

Codec	Aliases	Languages
utf_32_le	UTF-32LE	todas linguagens
utf_16	U16, utf16	todas linguagens
utf_16_be	UTF-16BE	todas linguagens
utf_16_le	UTF-16LE	todas linguagens
utf_7	U7, unicode-1-1-utf-7	todas linguagens
utf_8	U8, UTF, utf8	todas linguagens
utf_8_sig		todas linguagens

Alterado na versão 3.4: The utf-16* and utf-32* encoders no longer allow surrogate code points (`U+D800`–`U+DFFF`) to be encoded. The utf-32* decoders no longer decode byte sequences that correspond to surrogate code points.

7.2.4 Python Specific Encodings

A number of predefined codecs are specific to Python, so their codec names have no meaning outside Python. These are listed in the tables below based on the expected input and output types (note that while text encodings are the most common use case for codecs, the underlying codec infrastructure supports arbitrary data transforms rather than just text encodings). For asymmetric codecs, the stated meaning describes the encoding direction.

Text Encodings

The following codecs provide `str` to `bytes` encoding and *bytes-like object* to `str` decoding, similar to the Unicode text encodings.

Codec	Aliases	Significado
idna		Implement **RFC 3490**, see also *encodings.idna*. Only `errors='strict'` is supported.
mbcs	ansi, dbcs	Windows only: Encode the operand according to the ANSI codepage (CP_ACP).
oem		Windows only: Encode the operand according to the OEM codepage (CP_OEMCP). Novo na versão 3.6.
palmos		Encoding of PalmOS 3.5.
punycode		Implement **RFC 3492**. Stateful codecs are not supported.
raw_unicode_escape		Latin-1 encoding with `\uXXXX` and `\UXXXXXXXX` for other code points. Existing backslashes are not escaped in any way. It is used in the Python pickle protocol.
undefined		Raise an exception for all conversions, even empty strings. The error handler is ignored.
unicode_escape		Encoding suitable as the contents of a Unicode literal in ASCII-encoded Python source code, except that quotes are not escaped. Decode from Latin-1 source code. Beware that Python source code actually uses UTF-8 by default.
unicode_internal		Return the internal representation of the operand. Stateful codecs are not supported. Obsoleto desde a versão 3.3: This representation is obsoleted by **PEP 393**.

Binary Transforms

The following codecs provide binary transforms: *bytes-like object* to *bytes* mappings. They are not supported by *bytes.decode()* (which only produces *str* output).

Codec	Aliases	Significado	Encoder / decoder
base64_codec[1]	base64, base_64	Convert the operand to multiline MIME base64 (the result always includes a trailing `'\n'`). Alterado na versão 3.4: accepts any *bytes-like object* as input for encoding and decoding	`base64.encodebytes()` / `base64.decodebytes()`
bz2_codec	bz2	Compress the operand using bz2.	`bz2.compress()` / `bz2.decompress()`
hex_codec	hex	Convert the operand to hexadecimal representation, with two digits per byte.	`binascii.b2a_hex()` / `binascii.a2b_hex()`
quopri_codec	quopri, quotedprintable, quoted_printable	Convert the operand to MIME quoted printable.	`quopri.encode()` with `quotetabs=True` / `quopri.decode()`
uu_codec	uu	Convert the operand using uuencode.	`uu.encode()` / `uu.decode()`
zlib_codec	zip, zlib	Compress the operand using gzip.	`zlib.compress()` / `zlib.decompress()`

Novo na versão 3.2: Restoration of the binary transforms.

Alterado na versão 3.4: Restoration of the aliases for the binary transforms.

Text Transforms

The following codec provides a text transform: a `str` to `str` mapping. It is not supported by `str.encode()` (which only produces `bytes` output).

Codec	Aliases	Significado
rot_13	rot13	Return the Caesar-cypher encryption of the operand.

Novo na versão 3.2: Restoration of the `rot_13` text transform.

Alterado na versão 3.4: Restoration of the `rot13` alias.

7.2.5 `encodings.idna` — Internationalized Domain Names in Applications

This module implements **RFC 3490** (Internationalized Domain Names in Applications) and **RFC 3492** (Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)). It builds upon the `punycode` encoding and `stringprep`.

These RFCs together define a protocol to support non-ASCII characters in domain names. A domain name containing non-ASCII characters (such as `www.Alliancefrançaise.nu`) is converted into an ASCII-compatible encoding (ACE, such as `www.xn--alliancefranaise-npb.nu`). The ACE form of the domain name is then used in all places where arbitrary characters are not allowed by the protocol, such as DNS queries, HTTP `Host` fields, and so on. This conversion is carried out in the application; if possible invisible to the user: The application should transparently convert Unicode domain labels to IDNA on the wire, and convert back ACE labels to Unicode before presenting them to the user.

[1] In addition to *bytes-like objects*, `'base64_codec'` also accepts ASCII-only instances of `str` for decoding

Python supports this conversion in several ways: the `idna` codec performs conversion between Unicode and ACE, separating an input string into labels based on the separator characters defined in **section 3.1 of RFC 3490** and converting each label to ACE as required, and conversely separating an input byte string into labels based on the `.` separator and converting any ACE labels found into unicode. Furthermore, the *socket* module transparently converts Unicode host names to ACE, so that applications need not be concerned about converting host names themselves when they pass them to the socket module. On top of that, modules that have host names as function parameters, such as *http.client* and *ftplib*, accept Unicode host names (*http.client* then also transparently sends an IDNA hostname in the *Host* field if it sends that field at all).

When receiving host names from the wire (such as in reverse name lookup), no automatic conversion to Unicode is performed: applications wishing to present such host names to the user should decode them to Unicode.

The module *encodings.idna* also implements the nameprep procedure, which performs certain normalizations on host names, to achieve case-insensitivity of international domain names, and to unify similar characters. The nameprep functions can be used directly if desired.

`encodings.idna.`**`nameprep`**(*label*)
Return the nameprepped version of *label*. The implementation currently assumes query strings, so `AllowUnassigned` is true.

`encodings.idna.`**`ToASCII`**(*label*)
Convert a label to ASCII, as specified in **RFC 3490**. `UseSTD3ASCIIRules` is assumed to be false.

`encodings.idna.`**`ToUnicode`**(*label*)
Convert a label to Unicode, as specified in **RFC 3490**.

7.2.6 `encodings.mbcs` — Windows ANSI codepage

This module implements the ANSI codepage (CP_ACP).

Availability: Windows only.

Alterado na versão 3.3: Support any error handler.

Alterado na versão 3.2: Before 3.2, the *errors* argument was ignored; `'replace'` was always used to encode, and `'ignore'` to decode.

7.2.7 `encodings.utf_8_sig` — UTF-8 codec with BOM signature

This module implements a variant of the UTF-8 codec. On encoding, a UTF-8 encoded BOM will be prepended to the UTF-8 encoded bytes. For the stateful encoder this is only done once (on the first write to the byte stream). On decoding, an optional UTF-8 encoded BOM at the start of the data will be skipped.

CAPÍTULO 8

Tipos de Dados

Os módulos descritos neste capítulo fornecem uma variedade de tipos de dados especializados, como datas e horas, vetores de tipo fixo, filas de heap, filas de extremidade dupla e enumerações.

O Python também fornece alguns tipos de dados embutidos, em especial `dict`, `list`, `set` e `frozenset` e `tuple`. A classe `str` é usada para armazenar strings Unicode, e as classes `bytes` e `bytearray` são usadas para armazenar dados binários.

Os seguintes módulos estão documentados neste capítulo:

8.1 `datetime` — Tipos básicos de data e hora

Código Fonte: Lib/datetime.py

O módulo `datetime` fornece classes para manipular datas e tempo de forma simples ou complexas. Apesar de cálculos aritméticos com data e tempo serem suportados, o foco da implementação está na extração eficiente de atributo para formatar resultados e manipulação. Para funcionalidades relacionadas, veja também os módulos `time` e `calendar`.

There are two kinds of date and time objects: "naive" and "aware".

An aware object has sufficient knowledge of applicable algorithmic and political time adjustments, such as time zone and daylight saving time information, to locate itself relative to other aware objects. An aware object is used to represent a specific moment in time that is not open to interpretation[1].

A naive object does not contain enough information to unambiguously locate itself relative to other date/time objects. Whether a naive object represents Coordinated Universal Time (UTC), local time, or time in some other timezone is purely up to the program, just like it is up to the program whether a particular number represents metres, miles, or mass. Naive objects are easy to understand and to work with, at the cost of ignoring some aspects of reality.

For applications requiring aware objects, `datetime` and `time` objects have an optional time zone information attribute, `tzinfo`, that can be set to an instance of a subclass of the abstract `tzinfo` class. These `tzinfo` objects capture information about the offset from UTC time, the time zone name, and whether Daylight Saving Time is in effect. Note

[1] Se, isto é, nós ignoramos os efeitos da Relatividade

that only one concrete *tzinfo* class, the *timezone* class, is supplied by the *datetime* module. The *timezone* class can represent simple timezones with fixed offset from UTC, such as UTC itself or North American EST and EDT timezones. Supporting timezones at deeper levels of detail is up to the application. The rules for time adjustment across the world are more political than rational, change frequently, and there is no standard suitable for every application aside from UTC.

O módulo *datetime* exporta as seguintes constantes:

datetime.**MINYEAR**
O menor número de ano permitido no objeto :classe‘date‘ ou :classe‘.datetime‘ . :const‘MINYEAR‘ é `1`.

datetime.**MAXYEAR**
O maior número de ano permitido no objeto :classe‘date‘ ou :classe‘.datetime‘ . :const‘MAXYER‘ é `9999`.

Ver também:

Módulo *calendar* Funções gerais relacionadas ao calendário.

Módulo *time* Acesso de hora e conversões.

8.1.1 Tipos disponíveis

class datetime.**date**
Uma data ingênua idealizada, assumindo que o atual calendário Gregoriano sempre foi, e sempre estará em vigor. Atributos: *year*, *month*, e *day*.

class datetime.**time**
An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds (there is no notion of “leap seconds” here). Attributes: *hour*, *minute*, *second*, *microsecond*, and *tzinfo*.

class datetime.**datetime**
Uma combinação de uma data e uma hora. Atributos: `ano`, `mês`, `dia`, `hora`, `minuto`, `segundo`, `microsegundo`, e *tzinfo*.

class datetime.**timedelta**
Uma duração que expressa a diferença entre duas instâncias *date*, *time* ou *datetime* para resolução de microssegundos.

class datetime.**tzinfo**
An abstract base class for time zone information objects. These are used by the *datetime* and *time* classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).

class datetime.**timezone**
Uma classe que implementa a classe base abstrata **:classe:‘tzinfo‘** como um deslocamento fixo do UTC.

Novo na versão 3.2.

Objetos desses tipos são imutáveis.

Objetos do tipo: classe‘date‘ são sempre ingênuos.

An object of type *time* or *datetime* may be naive or aware. A *datetime* object *d* is aware if `d.tzinfo` is not `None` and `d.tzinfo.utcoffset(d)` does not return `None`. If `d.tzinfo` is `None`, or if `d.tzinfo` is not `None` but `d.tzinfo.utcoffset(d)` returns `None`, *d* is naive. A *time* object *t* is aware if `t.tzinfo` is not `None` and `t.tzinfo.utcoffset(None)` does not return `None`. Otherwise, *t* is naive.

The distinction between naive and aware doesn't apply to *timedelta* objects.

Relacionamentos de subclasses:

```
object
    timedelta
    tzinfo
        timezone
    time
    date
        datetime
```

8.1.2 `timedelta` Objetos

O objeto *timedelta* representa uma duração, a diferença entre duas datas ou horas.

class datetime.**timedelta**(*days=0*, *seconds=0*, *microseconds=0*, *milliseconds=0*, *minutes=0*, *hours=0*, *weeks=0*)

All arguments are optional and default to `0`. Arguments may be integers or floats, and may be positive or negative.

Only *days*, *seconds* and *microseconds* are stored internally. Arguments are converted to those units:

- Um milissegundo é convertido em 1000 microssegundos.
- Um minuto é convertido em 60 segundos.
- Uma hora é convertida em 3600 segundos.
- Uma semana é convertida para 7 dias.

e dias, segundos e microssegundos são normalizados para que a representação seja única, com

- `0 <= microsegundos < 1000000`
- `0 <= seconds < 3600*24` (o número de segundos em um dia)
- `-999999999 <= dias <= 999999999`

If any argument is a float and there are fractional microseconds, the fractional microseconds left over from all arguments are combined and their sum is rounded to the nearest microsecond using round-half-to-even tiebreaker. If no argument is a float, the conversion and normalization processes are exact (no information is lost).

Se o valor normalizado de dias estiver fora do intervalo indicado, *OverflowError* é gerado.

Note that normalization of negative values may be surprising at first. For example,

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

Class attributes are:

timedelta.**min**
: O mais negativo objeto *timedelta*, `timedelta(-999999999)`.

timedelta.**max**
: O mais positivo objeto *timedelta*, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

timedelta.**resolution**
: A menor diferença possível entre objetos não iguais *timedelta*, `timedelta(microseconds=1)`.

Observe que, devido à normalização, `timedelta.max` > `-timedelta.min`. `-timedelta.max` não é representável como um objeto *timedelta*.

Atributos de instância (somente leitura):

Atributo	Valor
`days`	Entre -999999999 e 999999999 inclusive
`seconds`	Entre 0 e 86399 inclusive
`microseconds`	Entre 0 e 999999 inclusive

Operações suportadas:

Operação	Resultado
`t1 = t2 + t3`	Soma de *t2* e *t3*. Depois *t1-t2 == t3* e *t1-t3 == t2* são verdadeiros. (1)
`t1 = t2 - t3`	Diferença de *t2* e *t3*. Depois *t1 == t2 - t3* e *t2 == t1 + t3* são verdadeiros (1)(6)
`t1 = t2 * i` ou `t1 = i * t2`	Delta multiplicado por um número inteiro. Depois *t1* // i == *t2* é verdadeiro, desde que `i != 0`.
	Em geral, *t1* * i == *t1* * (i-1) + *t1* é verdadeiro. (1)
`t1 = t2 * f` or `t1 = f * t2`	Delta multiplicado por um float, ponto flutuante. O resultado é arredondado para o múltiplo mais próximo de timedelta.resolution usando a metade da metade para o par.
`f = t2 / t3`	Divisão (3) da duração total *t2* por unidade de intervalo *t3*. Retorna um objeto *float*.
`t1 = t2 / f` or `t1 = t2 / i`	Delta dividido por um float ou um int. O resultado é arredondado para o múltiplo mais próximo de timedelta.resolution usando a metade da metade para o par.
`t1 = t2 // i` or `t1 = t2 // t3`	The floor is computed and the remainder (if any) is thrown away. In the second case, an integer is returned. (3)
`t1 = t2 % t3`	O restante é calculado como um objeto *timedelta*. (3)
`q, r = divmod(t1, t2)`	Calcula o quociente e o restante: `q = t1 // t2` (3) e `r = t1% t2`. q é um número inteiro e r é um objeto *timedelta*.
`+t1`	Retorna um objeto *timedelta* com o mesmo valor. (2)
`-t1`	equivalente a *timedelta*(*-t1.days*, *-t1.seconds*, *-t1.microseconds*), e a *t1** -1. (1)(4)
`abs(t)`	equivalente a + *t* quando `t.days> = 0`, e a - *t* quando `t.days <0`. (2)
`str(t)`	Retorna uma string no formato `[D day[s], ][H]H:MM:SS[.UUUUUU]`, onde D é negativo para `t` negativo. (5)
`repr(t)`	Retorna uma representação em string do objeto *timedelta* como uma chamada do construtor com valores de atributos canônicos.

Notas:

(1) This is exact, but may overflow.

(2) This is exact, and cannot overflow.

(3) A divisão por 0 gera *ZeroDivisionError*.

(4) *-timedelta.max* não é representável como um objeto *timedelta*.

(5) String representations of *timedelta* objects are normalized similarly to their internal representation. This leads to somewhat unusual results for negative timedeltas. For example:

```
>>> timedelta(hours=-5)
datetime.timedelta(days=-1, seconds=68400)
>>> print(_)
-1 day, 19:00:00
```

(6) A expressão `t2 - t3` sempre será igual à expressão `t2 + (-t3)` exceto quando t3 for igual a `timedelta.max`; nesse caso, o primeiro produzirá um resultado enquanto o último transbordará.

In addition to the operations listed above *timedelta* objects support certain additions and subtractions with *date* and *datetime* objects (see below).

Alterado na versão 3.2: Floor division and true division of a *timedelta* object by another *timedelta* object are now supported, as are remainder operations and the *divmod()* function. True division and multiplication of a *timedelta* object by a *float* object are now supported.

Comparisons of *timedelta* objects are supported with the *timedelta* object representing the smaller duration considered to be the smaller timedelta. In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a *timedelta* object is compared to an object of a different type, *TypeError* is raised unless the comparison is == or !=. The latter cases return *False* or *True*, respectively.

timedelta objects are *hashable* (usable as dictionary keys), support efficient pickling, and in Boolean contexts, a *timedelta* object is considered to be true if and only if it isn't equal to `timedelta(0)`.

Métodos de instância:

`timedelta.`**`total_seconds`**`()`

Retorna o número total de segundos contidos na duração. Equivalente a `td / timedelta(seconds=1)`. Para unidades de intervalo diferentes de segundos, use a forma de divisão diretamente (por exemplo `td / timedelta(microseconds=1)`).

Observe que, em intervalos de tempo muito grandes (mais de 270 anos na maioria das plataformas), esse método perde a precisão de microssegundos.

Novo na versão 3.2.

Example usage:

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)  # adds up to 365 days
>>> year.total_seconds()
31536000.0
>>> year == another_year
True
>>> ten_years = 10 * year
>>> ten_years, ten_years.days // 365
(datetime.timedelta(days=3650), 10)
>>> nine_years = ten_years - year
>>> nine_years, nine_years.days // 365
(datetime.timedelta(days=3285), 9)
>>> three_years = nine_years // 3
>>> three_years, three_years.days // 365
(datetime.timedelta(days=1095), 3)
>>> abs(three_years - ten_years) == 2 * three_years + year
True
```

8.1.3 `date` Objetos

A *date* object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar indefinitely extended in both directions. January 1 of year 1 is called day number 1, January 2 of year 1 is called day number 2, and so on. This matches the definition of the "proleptic Gregorian" calendar in Dershowitz and Reingold's book Calendrical Calculations, where it's the base calendar for all computations. See the book for algorithms for converting between proleptic Gregorian ordinals and many other calendar systems.

class datetime.**date**(*year*, *month*, *day*)
All arguments are required. Arguments must be integers in the following ranges:

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= day <= number of days in the given month and year`

Se um argumento fora desses intervalos for fornecido, *ValueError* é levantado.

Outros construtores, todos os métodos de classe.

classmethod date.**today**()
Return the current local date. This is equivalent to `date.fromtimestamp(time.time())`.

classmethod date.**fromtimestamp**(*timestamp*)
Return the local date corresponding to the POSIX timestamp, such as is returned by *time.time()*. This may raise *OverflowError*, if the timestamp is out of the range of values supported by the platform C `localtime()` function, and *OSError* on `localtime()` failure. It's common for this to be restricted to years from 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by *fromtimestamp()*.

Alterado na versão 3.3: Levanta *OverflowError* ao invés de *ValueError* se o carimbo de data/hora estiver fora do intervalo de valores suportados pela plataforma C `localtime()` função. Levanta *OSError* ao invés de *ValueError* em falha de `localtime()` .

classmethod date.**fromordinal**(*ordinal*)
Return the date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. *ValueError* is raised unless `1 <= ordinal <= date.max.toordinal()`. For any date *d*, `date.fromordinal(d.toordinal()) == d`.

classmethod date.**fromisoformat**(*date_string*)
Return a *date* corresponding to a *date_string* in the format emitted by *date.isoformat()*. Specifically, this function supports strings in the format(s) `YYYY-MM-DD`.

> **Cuidado:** This does not support parsing arbitrary ISO 8601 strings - it is only intended as the inverse operation of *date.isoformat()*.

Novo na versão 3.7.

Atributos de classe:

date.**min**
A data representável mais antiga, `date(MINYEAR, 1, 1)`.

date.**max**
A data representável mais tardia, `date(MAXYEAR, 12, 31)`.

date.**resolution**
A menor diferença possível entre objetos date não iguais, `timedelta(days=1)`.

Atributos de instância (somente leitura):

`date.`**`year`**
Entre *MINYEAR* e *MAXYEAR* incluindo extremos.

`date.`**`month`**
Entre 1 e 12 incluindo extremos.

`date.`**`day`**
Entre 1 e o número de dias no mês especificado do ano especificado.

Operações suportadas:

Operação	Resultado
`date2 = date1 + timedelta`	*date2* is `timedelta.days` days removed from *date1*. (1)
`date2 = date1 - timedelta`	Computa *date2* de modo que `date2 + timedelta == date1`. (2)
`timedelta = date1 - date2`	(3)
`date1 < date2`	*date1* é considerada menor que *date2* quando *date1* precede *date2* no tempo. (4)

Notas:

(1) *date2* is moved forward in time if `timedelta.days > 0`, or backward if `timedelta.days < 0`. Afterward `date2 - date1 == timedelta.days`. `timedelta.seconds` and `timedelta.microseconds` are ignored. *OverflowError* is raised if `date2.year` would be smaller than *MINYEAR* or larger than *MAXYEAR*.

(2) `timedelta.seconds` e `timedelta.microseconds` são ignoradas.

(3) This is exact, and cannot overflow. timedelta.seconds and timedelta.microseconds are 0, and date2 + timedelta == date1 after.

(4) In other words, `date1 < date2` if and only if `date1.toordinal() < date2.toordinal()`. Date comparison raises *TypeError* if the other comparand isn't also a *date* object. However, `NotImplemented` is returned instead if the other comparand has a `timetuple()` attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a *date* object is compared to an object of a different type, *TypeError* is raised unless the comparison is == or `!=`. The latter cases return *False* or *True*, respectively.

Dates can be used as dictionary keys. In Boolean contexts, all *date* objects are considered to be true.

Métodos de instância:

`date.`**`replace`**(*year=self.year*, *month=self.month*, *day=self.day*)
Return a date with the same value, except for those parameters given new values by whichever keyword arguments are specified. For example, if `d == date(2002, 12, 31)`, then `d.replace(day=26) == date(2002, 12, 26)`.

`date.`**`timetuple`**`()`
Return a *time.struct_time* such as returned by *time.localtime()*. The hours, minutes and seconds are 0, and the DST flag is -1. `d.timetuple()` is equivalent to `time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))`, where `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` is the day number within the current year starting with `1` for January 1st.

`date.`**`toordinal`**`()`
Return the proleptic Gregorian ordinal of the date, where January 1 of year 1 has ordinal 1. For any *date* object *d*, `date.fromordinal(d.toordinal()) == d`.

date.**weekday**()
Retorna o dia da semana como um inteiro, onde Segunda é 0 e Domingo é 6. Por exemplo, `date(2002, 12, 4).weekday() == 2`, uma Quarta-feira. Veja também *isoweekday()*.

date.**isoweekday**()
Retorna o dia da semana como um inteiro, onde Segunda é 1 e Domingo é 7. Por exemplo, `date(2002, 12, 4).isoweekday() == 3`, uma Quarta-feira. Veja também *weekday()*, *isocalendar()*.

date.**isocalendar**()
Return a 3-tuple, (ISO year, ISO week number, ISO weekday).

The ISO calendar is a widely used variant of the Gregorian calendar. See https://www.staff.science.uu.nl/~gent0113/calendar/isocalendar.htm for a good explanation.

The ISO year consists of 52 or 53 full weeks, and where a week starts on a Monday and ends on a Sunday. The first week of an ISO year is the first (Gregorian) calendar week of a year containing a Thursday. This is called week number 1, and the ISO year of that Thursday is the same as its Gregorian year.

For example, 2004 begins on a Thursday, so the first week of ISO year 2004 begins on Monday, 29 Dec 2003 and ends on Sunday, 4 Jan 2004, so that `date(2003, 12, 29).isocalendar() == (2004, 1, 1)` and `date(2004, 1, 4).isocalendar() == (2004, 1, 7)`.

date.**isoformat**()
Return a string representing the date in ISO 8601 format, 'YYYY-MM-DD'. For example, `date(2002, 12, 4).isoformat() == '2002-12-04'`.

date.**__str__**()
Para um objeto *date* *d*, `str(d)` é equivalente a `d.isoformat()`.

date.**ctime**()
Return a string representing the date, for example `date(2002, 12, 4).ctime() == 'Wed Dec 4 00:00:00 2002'`. `d.ctime()` is equivalent to `time.ctime(time.mktime(d.timetuple()))` on platforms where the native C `ctime()` function (which *time.ctime()* invokes, but which *date.ctime()* does not invoke) conforms to the C standard.

date.**strftime**(*format*)
Retorna uma string representando a data, controlado por uma string explícita de formatação. Códigos de formatação referenciando horas, minutos ou segundos irão ver valores 0. Para uma lista completa de diretivas de formatação, veja *Comportamento de strftime() e strptime()*.

date.**__format__**(*format*)
O mesmo que *date.strftime()*. Isso torna possível especificar uma string de formatação para o objeto *date* em literais de string formatados e ao usar *str.format()*. Para uma lista completa de diretivas de formatação, veja *Comportamento de strftime() e strptime()*.

Exemplo de contagem de dias para um evento:

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
```

(continua na próxima página)

(continuação da página anterior)

```
>>> time_to_birthday.days
202
```

Example of working with *date*:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)
>>> t = d.timetuple()
>>> for i in t:
...     print(i)
2002                # year
3                   # month
11                  # day
0
0
0
0                   # weekday (0 = Monday)
70                  # 70th day in the year
-1
>>> ic = d.isocalendar()
>>> for i in ic:
...     print(i)
2002                # ISO year
11                  # ISO week number
1                   # ISO day number ( 1 = Monday )
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'
```

8.1.4 Objetos `datetime`

A *datetime* object is a single object containing all the information from a *date* object and a *time* object. Like a *date* object, *datetime* assumes the current Gregorian calendar extended in both directions; like a time object, *datetime* assumes there are exactly 3600*24 seconds in every day.

Construtor:

class datetime.**datetime**(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)

The year, month and day arguments are required. *tzinfo* may be `None`, or an instance of a *tzinfo* subclass. The remaining arguments must be integers in the following ranges:

- `MINYEAR <= year <= MAXYEAR`,
- `1 <= month <= 12`,
- "1 <= day <= número de dias no mês e ano fornecidos',
- `0 <= hour < 24`,
- `0 <= minute < 60`,

- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold in [0, 1]`.

Se um argumento fora desses intervalos for fornecido, *ValueError* é levantado.

Novo na versão 3.6: Adicionado o argumento `fold`.

Outros construtores, todos os métodos de classe.

classmethod datetime.**today**()
Return the current local datetime, with *tzinfo* None. This is equivalent to `datetime.fromtimestamp(time.time())`. See also *now()*, *fromtimestamp()*.

classmethod datetime.**now**(*tz=None*)
Return the current local date and time. If optional argument *tz* is None or not specified, this is like *today()*, but, if possible, supplies more precision than can be gotten from going through a *time.time()* timestamp (for example, this may be possible on platforms supplying the C `gettimeofday()` function).

If *tz* is not None, it must be an instance of a *tzinfo* subclass, and the current date and time are converted to *tz*'s time zone. In this case the result is equivalent to `tz.fromutc(datetime.utcnow().replace(tzinfo=tz))`. See also *today()*, *utcnow()*.

classmethod datetime.**utcnow**()
Return the current UTC date and time, with *tzinfo* None. This is like *now()*, but returns the current UTC date and time, as a naive *datetime* object. An aware current UTC datetime can be obtained by calling `datetime.now(timezone.utc)`. See also *now()*.

classmethod datetime.**fromtimestamp**(*timestamp*, *tz=None*)
Retorna a data e hora local correspondente ao timestamp POSIX, como é retornado por *time.time()*. Se o argumento opcional *tz* é None ou não especificado, o timestamp é convertido para a data e hora local da plataforma, e o objeto *datetime* retornado é ingênuo.

If *tz* is not None, it must be an instance of a *tzinfo* subclass, and the timestamp is converted to *tz*'s time zone. In this case the result is equivalent to `tz.fromutc(datetime.utcfromtimestamp(timestamp).replace(tzinfo=tz))`.

fromtimestamp() may raise *OverflowError*, if the timestamp is out of the range of values supported by the platform C `localtime()` or `gmtime()` functions, and *OSError* on `localtime()` or `gmtime()` failure. It's common for this to be restricted to years in 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by *fromtimestamp()*, and then it's possible to have two timestamps differing by a second that yield identical *datetime* objects. See also *utcfromtimestamp()*.

Alterado na versão 3.3: Levanta um *OverflowError* ao invés de *ValueError* se o timestamp estiver fora do intervalo dos valores suportados pelas funções C `localtime()` ou `gmtime()` da plataforma. Levanta *OSError* ao invés de *ValueError* em falhas de `localtime()` ou `gmtime()`.

Alterado na versão 3.6: *fromtimestamp()* pode retornar instâncias com *fold* igual a 1.

classmethod datetime.**utcfromtimestamp**(*timestamp*)
Return the UTC *datetime* corresponding to the POSIX timestamp, with *tzinfo* None. This may raise *OverflowError*, if the timestamp is out of the range of values supported by the platform C `gmtime()` function, and *OSError* on `gmtime()` failure. It's common for this to be restricted to years in 1970 through 2038.

Para conseguir um objeto *datetime* consciente, chame `fromtiemstamp()`:

```
datetime.fromtimestamp(timestamp, timezone.utc)
```

Nas plataformas compatíveis com POSIX, é equivalente à seguinte expressão:

```
datetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta(seconds=timestamp)
```

com a exceção de que a última fórmula sempre dá suporte ao intervalo completo de anos: entre *MINYEAR* e *MAXYEAR* inclusive.

Alterado na versão 3.3: Levanta *OverflowError* ao invés de *ValueError* se o timestamp estiver fora do intervalo de valores suportados pela função C `gmtime()` da plataforma. Levanta *OSError* ao invés de *ValueError* em caso de falha `gmtime()`.

classmethod datetime.**fromordinal**(*ordinal*)

Return the *datetime* corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. *ValueError* is raised unless `1 <= ordinal <= datetime.max.toordinal()`. The hour, minute, second and microsecond of the result are all 0, and *tzinfo* is `None`.

classmethod datetime.**combine**(*date*, *time*, *tzinfo=self.tzinfo*)

Return a new *datetime* object whose date components are equal to the given *date* object's, and whose time components are equal to the given *time* object's. If the *tzinfo* argument is provided, its value is used to set the *tzinfo* attribute of the result, otherwise the *tzinfo* attribute of the *time* argument is used.

For any *datetime* object *d*, `d == datetime.combine(d.date(), d.time(), d.tzinfo)`. If date is a *datetime* object, its time components and *tzinfo* attributes are ignored.

Alterado na versão 3.6: Argumento *tzinfo* adicionado.

classmethod datetime.**fromisoformat**(*date_string*)

Return a *datetime* corresponding to a *date_string* in one of the formats emitted by *date.isoformat()* and *datetime.isoformat()*. Specifically, this function supports strings in the format(s) `YYYY-MM-DD[*HH[:MM[:SS[.fff[fff]]]][+HH:MM[:SS[.ffffff]]]]`, where `*` can match any single character.

Cuidado: This does not support parsing arbitrary ISO 8601 strings - it is only intended as the inverse operation of *datetime.isoformat()*. A more full-featured ISO 8601 parser, `dateutil.parser.isoparse` is available in the third-party package dateutil.

Novo na versão 3.7.

classmethod datetime.**strptime**(*date_string*, *format*)

Return a *datetime* corresponding to *date_string*, parsed according to *format*. This is equivalent to `datetime(*(time.strptime(date_string, format)[0:6]))`. *ValueError* is raised if the date_string and format can't be parsed by *time.strptime()* or if it returns a value which isn't a time tuple. For a complete list of formatting directives, see *Comportamento de strftime() e strptime()*.

Atributos de classe:

datetime.**min**

O primeiro *datetime* representável, `datetime(MINYEAR, 1, 1, tzinfo=None)`.

datetime.**max**

O último *datetime* representável, `datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)`.

datetime.**resolution**

A menor diferença possível entre objetos *datetime* desiguais, `timedelta(microseconds=1)`.

Atributos de instância (somente leitura):

`datetime.`**`year`**
Entre *MINYEAR* e *MAXYEAR* incluindo extremos.

`datetime.`**`month`**
Entre 1 e 12 incluindo extremos.

`datetime.`**`day`**
Entre 1 e o número de dias no mês especificado do ano especificado.

`datetime.`**`hour`**
No intervalo `range(24)`.

`datetime.`**`minute`**
No intervalo `range(60)`.

`datetime.`**`second`**
No intervalo `range(60)`.

`datetime.`**`microsecond`**
No intervalo `range(1000000)`.

`datetime.`**`tzinfo`**
O objeto passado como o argumento *tzinfo* do construtor *datetime*, ou `None` se nada foi passado.

`datetime.`**`fold`**
In `[0, 1]`. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The value 0 (1) represents the earlier (later) of the two moments with the same wall time representation.

Novo na versão 3.6.

Operações suportadas:

Operação	Resultado
`datetime2 = datetime1 + timedelta`	(1)
`datetime2 = datetime1 - timedelta`	(2)
`timedelta = datetime1 - datetime2`	(3)
`datetime1 < datetime2`	Compara um *datetime* a um *datetime*. (4)

(1) datetime2 is a duration of timedelta removed from datetime1, moving forward in time if `timedelta.days` > 0, or backward if `timedelta.days` < 0. The result has the same *tzinfo* attribute as the input datetime, and datetime2 - datetime1 == timedelta after. *OverflowError* is raised if datetime2.year would be smaller than *MINYEAR* or larger than *MAXYEAR*. Note that no time zone adjustments are done even if the input is an aware object.

(2) Computa o datetime2 tal que datetime2 + timedelta == datetime1. Assim como para adição, o resultado tem o mesmo atributo *tzinfo* que datetime de entrada, e nenhum ajuste de fuso horário é feito mesmo que a entrada seja consciente disso.

(3) Subtraction of a *datetime* from a *datetime* is defined only if both operands are naive, or if both are aware. If one is aware and the other is naive, *TypeError* is raised.

If both are naive, or both are aware and have the same *tzinfo* attribute, the *tzinfo* attributes are ignored, and the result is a *timedelta* object *t* such that `datetime2 + t == datetime1`. No time zone adjustments are done in this case.

If both are aware and have different *tzinfo* attributes, `a-b` acts as if *a* and *b* were first converted to naive UTC datetimes first. The result is `(a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())` except that the implementation never overflows.

(4) *datetime1* é considerado menor que *datetime2* quando *datetime1* precede *datetime2* no tempo.

If one comparand is naive and the other is aware, `TypeError` is raised if an order comparison is attempted. For equality comparisons, naive instances are never equal to aware instances.

If both comparands are aware, and have the same `tzinfo` attribute, the common `tzinfo` attribute is ignored and the base datetimes are compared. If both comparands are aware and have different `tzinfo` attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from `self.utcoffset()`).

Alterado na versão 3.3: Equality comparisons between naive and aware `datetime` instances don't raise `TypeError`.

Nota: In order to stop comparison from falling back to the default scheme of comparing object addresses, datetime comparison normally raises `TypeError` if the other comparand isn't also a `datetime` object. However, `NotImplemented` is returned instead if the other comparand has a `timetuple()` attribute. This hook gives other kinds of date objects a chance at implementing mixed-type comparison. If not, when a `datetime` object is compared to an object of a different type, `TypeError` is raised unless the comparison is == or `!=`. The latter cases return `False` or `True`, respectively.

`datetime` objects can be used as dictionary keys. In Boolean contexts, all `datetime` objects are considered to be true.

Métodos de instância:

`datetime.`**`date`**`()`
Retorna um objeto `date` com o mesmo ano, mês e dia.

`datetime.`**`time`**`()`
Return `time` object with same hour, minute, second, microsecond and fold. `tzinfo` is `None`. See also method `timetz()`.

Alterado na versão 3.6: O valor attr:.*fold* é copiado ao objeto `time` retornado.

`datetime.`**`timetz`**`()`
Return `time` object with same hour, minute, second, microsecond, fold, and tzinfo attributes. See also method `time()`.

Alterado na versão 3.6: O valor attr:.*fold* é copiado ao objeto `time` retornado.

`datetime.`**`replace`**`(`*year=self.year*, *month=self.month*, *day=self.day*, *hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, * *fold=0*`)`
Return a datetime with the same attributes, except for those attributes given new values by whichever keyword arguments are specified. Note that `tzinfo=None` can be specified to create a naive datetime from an aware datetime with no conversion of date and time data.

Novo na versão 3.6: Adicionado o argumento `fold`.

`datetime.`**`astimezone`**`(`*tz=None*`)`
Retorna um objeto `datetime` com novo atributo `tzinfo` definido por *tz*, ajustando a data e hora de forma que o resultado seja o mesmo horário UTC que *self*, mas na hora local de *tz*.

If provided, *tz* must be an instance of a `tzinfo` subclass, and its `utcoffset()` and `dst()` methods must not return `None`. If *self* is naive, it is presumed to represent time in the system timezone.

If called without arguments (or with `tz=None`) the system local timezone is assumed for the target timezone. The `.tzinfo` attribute of the converted datetime instance will be set to an instance of `timezone` with the zone name and offset obtained from the OS.

Se `self.tzinfo` for *tz*, `self.astimezone(tz)` é igual a *self*: nenhum ajuste nos dados de data ou hora é realizado. Caso contrário o resultado é a hora local no fuso horário *tz*, representando a mesma hora UTC que *self*:

depois `astz = dt.astimezone(tz)`, `astz - astz.utcoffset()` terá os mesmos dados de data e hora que `dt - dt.utcoffset()`.

If you merely want to attach a time zone object *tz* to a datetime *dt* without adjustment of date and time data, use `dt.replace(tzinfo=tz)`. If you merely want to remove the time zone object from an aware datetime *dt* without conversion of date and time data, use `dt.replace(tzinfo=None)`.

Perceba que o método padrão *tzinfo.fromutc()* pode ser substituído em uma subclasse *tzinfo* para afetar o resultado retornado por *astimezone()*. Ignorando erros de letras maiúsculas/minúsculas, *astimezone()* funciona como:

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new time zone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

Alterado na versão 3.3: *tz* agora pode ser omitido.

Alterado na versão 3.6: O método *astimezone()* agora pode ser chamado em instâncias ingênuas que presumidamente representam a hora local do sistema.

`datetime.`**`utcoffset`**`()`
Se *tzinfo* for `None`, retorna `None`, caso contrário retorna `self.tzinfo.utcoffset(self)`, e levanta uma exceção se o segundo não retornar `None` ou um objeto *timedelta* com magnitude menor que um dia.

Alterado na versão 3.7: A diferença UTC não é restrita a um número completo de minutos.

`datetime.`**`dst`**`()`
Se *tzinfo* for `None`, retorna `None`, caso contrário retorna `self.tzinfo.dst(self)`, e levanta uma exceção se o segundo não retornar `None` ou um objeto *timedelta* com magnitude menor que um dia.

Alterado na versão 3.7: A diferença de horário de verão não é restrita a um número completo de minutos.

`datetime.`**`tzname`**`()`
Se *tzinfo* for `None`, retorna `None`, caso contrário retorna `self.tzinfo.tzname(self)`, levanta uma exceção se o segundo não retornar `None` ou um objeto string.

`datetime.`**`timetuple`**`()`
Return a *time.struct_time* such as returned by *time.localtime()*. `d.timetuple()` is equivalent to `time.struct_time((d.year, d.month, d.day, d.hour, d.minute, d.second, d.weekday(), yday, dst))`, where `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` is the day number within the current year starting with `1` for January 1st. The `tm_isdst` flag of the result is set according to the *dst()* method: *tzinfo* is `None` or *dst()* returns `None`, `tm_isdst` is set to `-1`; else if *dst()* returns a non-zero value, `tm_isdst` is set to `1`; else `tm_isdst` is set to `0`.

`datetime.`**`utctimetuple`**`()`
If *datetime* instance *d* is naive, this is the same as `d.timetuple()` except that `tm_isdst` is forced to 0 regardless of what `d.dst()` returns. DST is never in effect for a UTC time.

If *d* is aware, *d* is normalized to UTC time, by subtracting `d.utcoffset()`, and a *time.struct_time* for the normalized time is returned. `tm_isdst` is forced to 0. Note that an *OverflowError* may be raised if *d*.year was `MINYEAR` or `MAXYEAR` and UTC adjustment spills over a year boundary.

`datetime.`**`toordinal`**`()`
Return the proleptic Gregorian ordinal of the date. The same as `self.date().toordinal()`.

datetime.**timestamp**()
Return POSIX timestamp corresponding to the *datetime* instance. The return value is a *float* similar to that returned by *time.time()*.

Naive *datetime* instances are assumed to represent local time and this method relies on the platform C `mktime()` function to perform the conversion. Since *datetime* supports wider range of values than `mktime()` on many platforms, this method may raise *OverflowError* for times far in the past or far in the future.

Para instâncias conscientes de *datetime*, o valor retornado é computado como:

```
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

Novo na versão 3.3.

Alterado na versão 3.6: O método *timestamp()* usa o atributo *fold* para desambiguar os tempos durante um intervalo repetido.

Nota: There is no method to obtain the POSIX timestamp directly from a naive *datetime* instance representing UTC time. If your application uses this convention and your system timezone is not set to UTC, you can obtain the POSIX timestamp by supplying `tzinfo=timezone.utc`:

```
timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
```

ou calculando o timestamp diretamente:

```
timestamp = (dt - datetime(1970, 1, 1)) / timedelta(seconds=1)
```

---

datetime.**weekday**()
Retorna o dia da semana como um inteiro, em que segunda-feira é 0 e domingo é 6. O mesmo que `self.date().weekday()`. Veja também *isoweekday()*.

datetime.**isoweekday**()
Retorna o dia da semana como um inteiro, em que segunda-feira é 1 e domingo é 7. O mesmo que `self.date().isoweekday()`. Veja também *weekday()*, *isocalendar()*.

datetime.**isocalendar**()
Return a 3-tuple, (ISO year, ISO week number, ISO weekday). The same as `self.date().isocalendar()`.

datetime.**isoformat**(*sep='T'*, *timespec='auto'*)
Return a string representing the date and time in ISO 8601 format, YYYY-MM-DDTHH:MM:SS.ffffff or, if *microsecond* is 0, YYYY-MM-DDTHH:MM:SS

If *utcoffset()* does not return `None`, a string is appended, giving the UTC offset: YYYY-MM-DDTHH:MM:SS.ffffff+HH:MM[:SS[.ffffff]] or, if *microsecond* is 0 YYYY-MM-DDTHH:MM:SS+HH:MM[:SS[.ffffff]].

The optional argument *sep* (default `'T'`) is a one-character separator, placed between the date and time portions of the result. For example,

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     def utcoffset(self, dt): return timedelta(minutes=-399)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
```

O argumento opcional *timespec* especifica o número de componentes adicionais do tempo a incluir (o padrão é `'auto'`). Pode ser uma das seguintes strings:

- `'auto'`: O mesmo que `'seconds'` se *microsecond* é 0, o mesmo que `'microseconds'` caso contrário.
- `'hours'`: Include the *hour* in the two-digit HH format.
- `'minutes'`: Include *hour* and *minute* in HH:MM format.
- `'seconds'`: Include *hour*, *minute*, and *second* in HH:MM:SS format.
- `'milliseconds'`: Include full time, but truncate fractional second part to milliseconds. HH:MM:SS.sss format.
- `'microseconds'`: Include full time in HH:MM:SS.ffffff format.

Nota: Componentes do tempo excluídos são truncados, não arredondados.

ValueError será levantado com um argumento *timespec* inválido.

```
>>> from datetime import datetime
>>> datetime.now().isoformat(timespec='minutes')
'2002-12-25T00:00'
>>> dt = datetime(2015, 1, 1, 12, 30, 59, 0)
>>> dt.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

Novo na versão 3.6: Argumento *timespec* adicionado.

`datetime.__str__()`
Para uma instância *datetime* *d*, `str(d)` é equivalente a `d.isoformat(' ')`.

`datetime.ctime()`
Return a string representing the date and time, for example `datetime(2002, 12, 4, 20, 30, 40).ctime() == 'Wed Dec 4 20:30:40 2002'`. `d.ctime()` is equivalent to `time.ctime(time.mktime(d.timetuple()))` on platforms where the native C `ctime()` function (which *time.ctime()* invokes, but which *datetime.ctime()* does not invoke) conforms to the C standard.

`datetime.strftime(`*format*`)`
Return a string representing the date and time, controlled by an explicit format string. For a complete list of formatting directives, see *Comportamento de strftime() e strptime()*.

`datetime.__format__(`*format*`)`
Same as *datetime.strftime()*. This makes it possible to specify a format string for a *datetime* object in formatted string literals and when using *str.format()*. For a complete list of formatting directives, see *Comportamento de strftime() e strptime()*.

Examples of working with datetime objects:

```
>>> from datetime import datetime, date, time
>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)
>>> # Using datetime.now() or datetime.utcnow()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
```

(continua na próxima página)

(continuação da página anterior)

```
>>> datetime.utcnow()
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060)
>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)
>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print(it)
...
2006    # year
11      # month
21      # day
16      # hour
30      # minute
0       # second
1       # weekday (0 = Monday)
325     # number of days since 1st January
-1      # dst - method tzinfo.dst() returned None
>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print(it)
...
2006    # ISO year
47      # ISO week
2       # ISO weekday
>>> # Formatting datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.'.format(dt, "day",
↪"month", "time")
'The day is 21, the month is November, the time is 04:30PM.'
```

Using datetime with tzinfo:

```
>>> from datetime import timedelta, datetime, tzinfo, timezone
>>> class KabulTz(tzinfo):
...     # Kabul used +4 until 1945, when they moved to +4:30
...     UTC_MOVE_DATE = datetime(1944, 12, 31, 20, tzinfo=timezone.utc)
...     def utcoffset(self, dt):
...         if dt.year < 1945:
...             return timedelta(hours=4)
...         elif (1945, 1, 1, 0, 0) <= dt.timetuple()[:5] < (1945, 1, 1, 0, 30):
...             # If dt falls in the imaginary range, use fold to decide how
...             # to resolve. See PEP495
...             return timedelta(hours=4, minutes=(30 if dt.fold else 0))
...         else:
...             return timedelta(hours=4, minutes=30)
...
...     def fromutc(self, dt):
...         # A custom implementation is required for fromutc as
...         # the input to this function is a datetime with utc values
...         # but with a tzinfo set to self
...         # See datetime.astimezone or fromtimestamp
...
```

(continua na próxima página)

(continuação da página anterior)

```
...         # Follow same validations as in datetime.tzinfo
...         if not isinstance(dt, datetime):
...             raise TypeError("fromutc() requires a datetime argument")
...         if dt.tzinfo is not self:
...             raise ValueError("dt.tzinfo is not self")
...
...         if dt.replace(tzinfo=timezone.utc) >= self.UTC_MOVE_DATE:
...             return dt + timedelta(hours=4, minutes=30)
...         else:
...             return dt + timedelta(hours=4)
...
...     def dst(self, dt):
...         return timedelta(0)
...
...     def tzname(self, dt):
...         if dt >= self.UTC_MOVE_DATE:
...             return "+04:30"
...         else:
...             return "+04"
...
...     def  __repr__(self):
...         return f"{self.__class__.__name__}()"
...
>>> tz1 = KabulTz()
>>> # Datetime before the change
>>> dt1 = datetime(1900, 11, 21, 16, 30, tzinfo=tz1)
>>> print(dt1.utcoffset())
4:00:00
>>> # Datetime after the change
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=tz1)
>>> print(dt2.utcoffset())
4:30:00
>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(timezone.utc)
>>> dt3
datetime.datetime(2006, 6, 14, 8, 30, tzinfo=datetime.timezone.utc)
>>> dt2
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=KabulTz())
>>> dt2.utctimetuple() == dt3.utctimetuple()
True
```

8.1.5 Objetos `time`

A time object represents a (local) time of day, independent of any particular day, and subject to adjustment via a *tzinfo* object.

class datetime.**time**(*hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)

All arguments are optional. *tzinfo* may be `None`, or an instance of a *tzinfo* subclass. The remaining arguments must be integers in the following ranges:

- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,

- `fold in [0, 1]`.

If an argument outside those ranges is given, *ValueError* is raised. All default to `0` except *tzinfo*, which defaults to *None*.

Atributos de classe:

`time.`**min**
O *time* mais cedo que pode ser representado, `time(0, 0, 0, 0)`.

`time.`**max**
O *time* mais tardio que pode ser representado, `time(23, 59, 59, 999999)`.

`time.`**resolution**
A menor diferença possível entre objetos *time* diferentes, `timedelta(microseconds=1)`, embora perceba que aritmética sobre objetos *time* não é suportada.

Atributos de instância (somente leitura):

`time.`**hour**
No intervalo `range(24)`.

`time.`**minute**
No intervalo `range(60)`.

`time.`**second**
No intervalo `range(60)`.

`time.`**microsecond**
No intervalo `range(1000000)`.

`time.`**tzinfo**
O objeto passado como argumento tzinfo para o construtor da classe *time*, ou `None` se nada foi passado.

`time.`**fold**
In `[0, 1]`. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The value 0 (1) represents the earlier (later) of the two moments with the same wall time representation.

Novo na versão 3.6.

Operações suportadas:

- comparison of *time* to *time*, where *a* is considered less than *b* when *a* precedes *b* in time. If one comparand is naive and the other is aware, *TypeError* is raised if an order comparison is attempted. For equality comparisons, naive instances are never equal to aware instances.

  If both comparands are aware, and have the same *tzinfo* attribute, the common *tzinfo* attribute is ignored and the base times are compared. If both comparands are aware and have different *tzinfo* attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from `self.utcoffset()`). In order to stop mixed-type comparisons from falling back to the default comparison by object address, when a *time* object is compared to an object of a different type, *TypeError* is raised unless the comparison is `==` or `!=`. The latter cases return *False* or *True*, respectively.

  Alterado na versão 3.3: Equality comparisons between naive and aware *time* instances don't raise *TypeError*.

- hash, use as dict key
- efficient pickling

In boolean contexts, a *time* object is always considered to be true.

Alterado na versão 3.5: Before Python 3.5, a *time* object was considered to be false if it represented midnight in UTC. This behavior was considered obscure and error-prone and has been removed in Python 3.5. See bpo-13936 for full details.

Outro construtor:

classmethod time.**fromisoformat**(*time_string*)

Return a *time* corresponding to a *time_string* in one of the formats emitted by *time.isoformat()*. Specifically, this function supports strings in the format(s) `HH[:MM[:SS[.fff[fff]]]][+HH:MM[:SS[.ffffff]]]`.

> **Cuidado:** This does not support parsing arbitrary ISO 8601 strings - it is only intended as the inverse operation of *time.isoformat()*.

Novo na versão 3.7.

Métodos de instância:

time.**replace**(*hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, ** fold=0*)

Return a *time* with the same value, except for those attributes given new values by whichever keyword arguments are specified. Note that `tzinfo=None` can be specified to create a naive *time* from an aware *time*, without conversion of the time data.

Novo na versão 3.6: Adicionado o argumento `fold`.

time.**isoformat**(*timespec='auto'*)

Return a string representing the time in ISO 8601 format, HH:MM:SS.ffffff or, if *microsecond* is 0, HH:MM:SS If *utcoffset()* does not return `None`, a string is appended, giving the UTC offset: HH:MM:SS.ffffff+HH:MM[:SS[.ffffff]] or, if self.microsecond is 0, HH:MM:SS+HH:MM[:SS[.ffffff]].

O argumento opcional *timespec* especifica o número de componentes adicionais do tempo a incluir (o padrão é `'auto'`). Pode ser uma das seguintes strings:

- `'auto'`: O mesmo que `'seconds'` se *microsecond* é 0, o mesmo que `'microseconds'` caso contrário.
- `'hours'`: Include the *hour* in the two-digit HH format.
- `'minutes'`: Include *hour* and *minute* in HH:MM format.
- `'seconds'`: Include *hour*, *minute*, and *second* in HH:MM:SS format.
- `'milliseconds'`: Include full time, but truncate fractional second part to milliseconds. HH:MM:SS.sss format.
- `'microseconds'`: Include full time in HH:MM:SS.ffffff format.

> **Nota:** Componentes do tempo excluídos são truncados, não arredondados.

ValueError será levantado com um argumento *timespec* inválido.

```
>>> from datetime import time
>>> time(hour=12, minute=34, second=56, microsecond=123456).isoformat(timespec=
↪'minutes')
'12:34'
>>> dt = time(hour=12, minute=34, second=56, microsecond=0)
>>> dt.isoformat(timespec='microseconds')
```

(continua na próxima página)

(continuação da página anterior)

```
'12:34:56.000000'
>>> dt.isoformat(timespec='auto')
'12:34:56'
```

Novo na versão 3.6: Argumento *timespec* adicionado.

`time.__str__()`
Para um tempo *t*, `str(t)` é equivalente a `t.isoformat()`.

`time.strftime(format)`
Return a string representing the time, controlled by an explicit format string. For a complete list of formatting directives, see *Comportamento de strftime() e strptime()*.

`time.__format__(format)`
Same as *time.strftime()*. This makes it possible to specify a format string for a *time* object in formatted string literals and when using *str.format()*. For a complete list of formatting directives, see *Comportamento de strftime() e strptime()*.

`time.utcoffset()`
Se *tzinfo* for `None`, retorna `None`, caso contrário retorna `self.tzinfo.utcoffset(None)`, e levanta uma exceção se o segundo não retornar `None` ou um objeto *timedelta* com magnitude menor que um dia.

Alterado na versão 3.7: A diferença UTC não é restrita a um número completo de minutos.

`time.dst()`
Se *tzinfo* for `None`, retorna `None`, caso contrário retorna `self.tzinfo.dst(None)`, e levanta uma exceção se o segundo não retornar `None`, ou um objeto *timedelta* com magnitude menor que um dia.

Alterado na versão 3.7: A diferença de horário de verão não é restrita a um número completo de minutos.

`time.tzname()`
Se *tzinfo* for `None`, retorna `None`, caso contrário retorna `self.tzinfo.tzname(None)`, ou levanta uma exceção se o último caso não retornar `None` ou um objeto string.

Exemplo:

```
>>> from datetime import time, tzinfo, timedelta
>>> class TZ1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self,dt):
...         return "+01:00"
...     def  __repr__(self):
...         return f"{self.__class__.__name__}()"
...
>>> t = time(12, 10, 30, tzinfo=TZ1())
>>> t
datetime.time(12, 10, 30, tzinfo=TZ1())
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'+01:00'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 +01:00'
```

(continua na próxima página)

(continuação da página anterior)

```
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

8.1.6 Objetos `tzinfo`

class datetime.**tzinfo**

This is an abstract base class, meaning that this class should not be instantiated directly. You need to derive a concrete subclass, and (at least) supply implementations of the standard *tzinfo* methods needed by the *datetime* methods you use. The *datetime* module supplies a simple concrete subclass of *tzinfo*, *timezone*, which can represent timezones with fixed offset from UTC such as UTC itself or North American EST and EDT.

Uma instância de (uma subclasse concreta de) *tzinfo* pode ser passada para os construtores de objetos *datetime* e *time*. Os objetos time veem seus atributos como se estivessem em horário local, e o objeto *tzinfo* suporta métodos revelando a diferença da hora local a partir de UTC, o nome do fuso horário, e diferença de horário em horário de verão, todos relativos ao objeto date ou time passado para eles.

Special requirement for pickling: A *tzinfo* subclass must have an `__init__()` method that can be called with no arguments, else it can be pickled but possibly not unpickled again. This is a technical requirement that may be relaxed in the future.

A concrete subclass of *tzinfo* may need to implement the following methods. Exactly which methods are needed depends on the uses made of aware *datetime* objects. If in doubt, simply implement all of them.

tzinfo.**utcoffset**(*dt*)

Return offset of local time from UTC, as a *timedelta* object that is positive east of UTC. If local time is west of UTC, this should be negative. Note that this is intended to be the total offset from UTC; for example, if a *tzinfo* object represents both time zone and DST adjustments, *utcoffset()* should return their sum. If the UTC offset isn't known, return `None`. Else the value returned must be a *timedelta* object strictly between `-timedelta(hours=24)` and `timedelta(hours=24)` (the magnitude of the offset must be less than one day). Most implementations of *utcoffset()* will probably look like one of these two:

```
return CONSTANT                 # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware class
```

Se *utcoffset()* não retorna `None`, *dst()* também não deve retornar `None`.

A implementação padrão de *utcoffset()* levanta *NotImplementedError*.

Alterado na versão 3.7: A diferença UTC não é restrita a um número completo de minutos.

tzinfo.**dst**(*dt*)

Return the daylight saving time (DST) adjustment, as a *timedelta* object or `None` if DST information isn't known. Return `timedelta(0)` if DST is not in effect. If DST is in effect, return the offset as a *timedelta* object (see *utcoffset()* for details). Note that DST offset, if applicable, has already been added to the UTC offset returned by *utcoffset()*, so there's no need to consult *dst()* unless you're interested in obtaining DST info separately. For example, *datetime.timetuple()* calls its *tzinfo* attribute's *dst()* method to determine how the `tm_isdst` flag should be set, and *tzinfo.fromutc()* calls *dst()* to account for DST changes when crossing time zones.

Uma instância *tz* de uma subclasse *tzinfo* que modela tanto horário padrão quanto horário de verão deve ser consistente neste sentido:

`tz.utcoffset(dt) - tz.dst(dt)`

must return the same result for every *datetime* *dt* with `dt.tzinfo == tz` For sane *tzinfo* subclasses, this expression yields the time zone's "standard offset", which should not depend on the date or the time, but only on geographic location. The implementation of *datetime.astimezone()* relies on this, but cannot detect

violations; it's the programmer's responsibility to ensure it. If a *tzinfo* subclass cannot guarantee this, it may be able to override the default implementation of *tzinfo.fromutc()* to work correctly with astimezone() regardless.

Maior parte das implementações de *dst()* provavelmente irá parecer com um destes dois:

```
def dst(self, dt):
    # a fixed-offset class:  doesn't account for DST
    return timedelta(0)
```

ou

```
def dst(self, dt):
    # Code to set dston and dstoff to the time zone's DST
    # transition times based on the input dt.year, and expressed
    # in standard local time.  Then

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

A implementação padrão de *dst()* levanta *NotImplementedError*.

Alterado na versão 3.7: A diferença de horário de verão não é restrita a um número completo de minutos.

tzinfo.**tzname**(*dt*)

Return the time zone name corresponding to the *datetime* object *dt*, as a string. Nothing about string names is defined by the *datetime* module, and there's no requirement that it mean anything in particular. For example, "GMT", "UTC", "-500", "-5:00", "EDT", "US/Eastern", "America/New York" are all valid replies. Return None if a string name isn't known. Note that this is a method rather than a fixed string primarily because some *tzinfo* subclasses will wish to return different names depending on the specific value of *dt* passed, especially if the *tzinfo* class is accounting for daylight time.

A implementação padrão de *tzname()* levanta *NotImplementedError*.

These methods are called by a *datetime* or *time* object, in response to their methods of the same names. A *datetime* object passes itself as the argument, and a *time* object passes None as the argument. A *tzinfo* subclass's methods should therefore be prepared to accept a *dt* argument of None, or of class *datetime*.

When None is passed, it's up to the class designer to decide the best response. For example, returning None is appropriate if the class wishes to say that time objects don't participate in the *tzinfo* protocols. It may be more useful for utcoffset(None) to return the standard UTC offset, as there is no other convention for discovering the standard offset.

When a *datetime* object is passed in response to a *datetime* method, dt.tzinfo is the same object as *self*. *tzinfo* methods can rely on this, unless user code calls *tzinfo* methods directly. The intent is that the *tzinfo* methods interpret *dt* as being in local time, and not need worry about objects in other timezones.

Exste mais um método *tzinfo* que uma subclasse pode querer substituir:

tzinfo.**fromutc**(*dt*)

This is called from the default *datetime.astimezone()* implementation. When called from that, dt.tzinfo is *self*, and *dt*'s date and time data are to be viewed as expressing a UTC time. The purpose of *fromutc()* is to adjust the date and time data, returning an equivalent datetime in *self*'s local time.

Most *tzinfo* subclasses should be able to inherit the default *fromutc()* implementation without problems. It's strong enough to handle fixed-offset time zones, and time zones accounting for both standard and daylight time, and the latter even if the DST transition times differ in different years. An example of a time zone the default *fromutc()* implementation may not handle correctly in all cases is one where the standard offset (from UTC)

depends on the specific date and time passed, which can happen for political reasons. The default implementations of `astimezone()` and *`fromutc()`* may not produce the result you want if the result is one of the hours straddling the moment the standard offset changes.

Ignorando o código para casos de erros, a implementação padrão *`fromutc()`* funciona como:

```
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst  # this is self's standard offset
    if delta:
        dt += delta   # convert to standard local time
        dtdst = dt.dst()
        # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

No seguinte arquivo `tzinfo_examples.py` existem alguns exemplos de classes *`tzinfo`*:

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
SECOND = timedelta(seconds=1)

# A class capturing the platform's idea of local time.
# (May result in wrong values on historical times in
#  timezones where UTC offset and/or the DST rules had
#  changed in the past.)
import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def fromutc(self, dt):
        assert dt.tzinfo is self
        stamp = (dt - datetime(1970, 1, 1, tzinfo=self)) // SECOND
        args = _time.localtime(stamp)[:6]
        dst_diff = DSTDIFF // SECOND
        # Detect fold
        fold = (args == _time.localtime(stamp - dst_diff))
        return datetime(*args, microsecond=dt.microsecond,
                        tzinfo=self, fold=fold)

    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
```

(continua na próxima página)

(continuação da página anterior)

```
        else:
            return STDOFFSET

    def dst(self, dt):
        if self._isdst(dt):
            return DSTDIFF
        else:
            return ZERO

    def tzname(self, dt):
        return _time.tzname[self._isdst(dt)]

    def _isdst(self, dt):
        tt = (dt.year, dt.month, dt.day,
              dt.hour, dt.minute, dt.second,
              dt.weekday(), 0, 0)
        stamp = _time.mktime(tt)
        tt = _time.localtime(stamp)
        return tt.tm_isdst > 0

Local = LocalTimezone()


# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt


# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases) set of rules for US
# DST start and end times. For a complete and up-to-date set of DST rules
# and timezone definitions, visit the Olson Database (or try pytz):
# http://www.twinsun.com/tz/tz-link.htm
# http://sourceforge.net/projects/pytz/ (might not be up-to-date)
#
# In the US, since 2007, DST starts at 2am (standard time) on the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 2)
# From 1987 to 2006, DST used to start at 2am (standard time) on the first
# Sunday in April and to end at 2am (DST time) on the last
# Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 2)
# From 1967 to 1986, DST used to start at 2am (standard time) on the last
# Sunday in April (the one on or after April 24) and to end at 2am (DST time)
# on the last Sunday of October, which is the first Sunday
# on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006
```

(continua na próxima página)

(continuação da página anterior)

```
def us_dst_range(year):
    # Find start and end times for US DST. For years before 1967, return
    # start = end for no DST.
    if 2006 < year:
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < year < 2007:
        dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
    elif 1966 < year < 1987:
        dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
    else:
        return (datetime(year, 1, 1), ) * 2

    start = first_sunday_on_or_after(dststart.replace(year=year))
    end = first_sunday_on_or_after(dstend.replace(year=year))
    return start, end


class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname

    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)

    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both cases.
            # It depends on how you want to treat them.  The default
            # fromutc() implementation (called by the default astimezone()
            # implementation) passes a datetime with dt.tzinfo is self.
            return ZERO
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        dt = dt.replace(tzinfo=None)
        if start + HOUR <= dt < end - HOUR:
            # DST is in effect.
            return HOUR
        if end - HOUR <= dt < end:
            # Fold (an ambiguous hour): use dt.fold to disambiguate.
            return ZERO if dt.fold else HOUR
        if start <= dt < start + HOUR:
            # Gap (a non-existent hour): reverse the fold rule.
```

(continua na próxima página)

(continuação da página anterior)

```
            return HOUR if dt.fold else ZERO
        # DST is off.
        return ZERO

    def fromutc(self, dt):
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        start = start.replace(tzinfo=self)
        end = end.replace(tzinfo=self)
        std_time = dt + self.stdoffset
        dst_time = std_time + HOUR
        if end <= dst_time < end + HOUR:
            # Repeated hour
            return std_time.replace(fold=1)
        if std_time < start or dst_time >= end:
            # Standard time
            return std_time
        if start <= std_time < end - HOUR:
            # Daylight saving time
            return dst_time


Eastern  = USTimeZone(-5, "Eastern",  "EST", "EDT")
Central  = USTimeZone(-6, "Central",  "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific  = USTimeZone(-8, "Pacific",  "PST", "PDT")
```

Note that there are unavoidable subtleties twice per year in a *tzinfo* subclass accounting for both standard and daylight time, at the DST transition points. For concreteness, consider US Eastern (UTC -0500), where EDT begins the minute after 1:59 (EST) on the second Sunday in March, and ends the minute after 1:59 (EDT) on the first Sunday in November:

```
  UTC   3:MM  4:MM  5:MM  6:MM  7:MM  8:MM
  EST  22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
  EDT  23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

start  22:MM 23:MM  0:MM  1:MM  3:MM  4:MM

  end  23:MM  0:MM  1:MM  1:MM  2:MM  3:MM
```

When DST starts (the "start" line), the local wall clock leaps from 1:59 to 3:00. A wall time of the form 2:MM doesn't really make sense on that day, so `astimezone(Eastern)` won't deliver a result with `hour == 2` on the day DST begins. For example, at the Spring forward transition of 2016, we get

```
>>> from datetime import datetime, timezone
>>> from tzinfo_examples import HOUR, Eastern
>>> u0 = datetime(2016, 3, 13, 5, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname())
...
05:00:00 UTC = 00:00:00 EST
06:00:00 UTC = 01:00:00 EST
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT
```

When DST ends (the "end" line), there's a potentially worse problem: there's an hour that can't be spelled unambiguously

in local wall time: the last hour of daylight time. In Eastern, that's times of the form 5:MM UTC on the day daylight time ends. The local wall clock leaps from 1:59 (daylight time) back to 1:00 (standard time) again. Local times of the form 1:MM are ambiguous. `astimezone()` mimics the local clock's behavior by mapping two adjacent UTC hours into the same local hour then. In the Eastern example, UTC times of the form 5:MM and 6:MM both map to 1:MM when converted to Eastern, but earlier times have the *fold* attribute set to 0 and the later times have it set to 1. For example, at the Fall back transition of 2016, we get

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

Note that the *datetime* instances that differ only by the value of the *fold* attribute are considered equal in comparisons.

Aplicações que não podem suportar horário na parede com ambiguidades devem explicitamente verificar o valor do atributo *fold* ou evitar o uso de subclasses *tzinfo* híbridas; não existem ambiguidades ao usar *timezone*, ou qualquer outra subclasse *tzinfo* com diferença fixa (tal como uma classe representando apenas o horário padrão na costa leste EST (diferença fixa de -5 horas), ou apenas o horário de verão na costa leste EDT (diferença fixa de -4 horas)).

Ver também:

dateutil.tz The standard library has *timezone* class for handling arbitrary fixed offsets from UTC and *timezone.utc* as UTC timezone instance.

dateutil.tz library brings the *IANA timezone database* (also known as the Olson database) to Python and its usage is recommended.

Base de dados de fusos horários IANA O banco de dados de fuso horário (comumente chando de tz, tzdata ou zoneinfo) contém código e dados que representam o histórico de hora local para muitas localizações representativas ao redor do globo. Ele é atualizado periodicamente para refletir mudanças feitas por corpos políticos para limites de fuso horário, diferenças UTC, e regras de horário de verão.

8.1.7 Objetos `timezone`

The *timezone* class is a subclass of *tzinfo*, each instance of which represents a timezone defined by a fixed offset from UTC. Note that objects of this class cannot be used to represent timezone information in the locations where different offsets are used in different days of the year or where historical changes have been made to civil time.

class datetime.**timezone**(*offset*, *name=None*)

The *offset* argument must be specified as a *timedelta* object representing the difference between the local time and UTC. It must be strictly between `-timedelta(hours=24)` and `timedelta(hours=24)`, otherwise *ValueError* is raised.

The *name* argument is optional. If specified it must be a string that will be used as the value returned by the *datetime.tzname()* method.

Novo na versão 3.2.

Alterado na versão 3.7: A diferença UTC não é restrita a um número completo de minutos.

timezone.**utcoffset**(*dt*)

Return the fixed value specified when the *timezone* instance is constructed. The *dt* argument is ignored. The return value is a *timedelta* instance equal to the difference between the local time and UTC.

Alterado na versão 3.7: A diferença UTC não é restrita a um número completo de minutos.

`timezone.`**`tzname`**(*dt*)
Return the fixed value specified when the *timezone* instance is constructed. If *name* is not provided in the constructor, the name returned by `tzname(dt)` is generated from the value of the `offset` as follows. If *offset* is `timedelta(0)`, the name is "UTC", otherwise it is a string 'UTC±HH:MM', where ± is the sign of `offset`, HH and MM are two digits of `offset.hours` and `offset.minutes` respectively.

Alterado na versão 3.6: Name generated from `offset=timedelta(0)` is now plain 'UTC', not 'UTC+00:00'.

`timezone.`**`dst`**(*dt*)
Sempre retorna `None`.

`timezone.`**`fromutc`**(*dt*)
Return `dt + offset`. The *dt* argument must be an aware *datetime* instance, with `tzinfo` set to `self`.

Atributos de classe:

`timezone.`**`utc`**
O fuso horário UTC, `timezone(timedelta(0))`.

8.1.8 Comportamento de `strftime()` e `strptime()`

date, *datetime*, and *time* objects all support a `strftime(format)` method, to create a string representing the time under the control of an explicit format string. Broadly speaking, `d.strftime(fmt)` acts like the *time* module's `time.strftime(fmt, d.timetuple())` although not all objects support a `timetuple()` method.

Conversely, the *datetime.strptime()* class method creates a *datetime* object from a string representing a date and time and a corresponding format string. `datetime.strptime(date_string, format)` is equivalent to `datetime(*(time.strptime(date_string, format)[0:6]))`, except when the format includes sub-second components or timezone offset information, which are supported in `datetime.strptime` but are discarded by `time.strptime`.

For *time* objects, the format codes for year, month, and day should not be used, as time objects have no such values. If they're used anyway, `1900` is substituted for the year, and `1` for the month and day.

For *date* objects, the format codes for hours, minutes, seconds, and microseconds should not be used, as *date* objects have no such values. If they're used anyway, `0` is substituted for them.

Para o método de classe *datetime.strptime()*, o valor padrão é `1900-01-01T00:00:00.000`: quaisquer componentes não especificados no formato da string serão puxados do valor padrão.[2]

The full set of format codes supported varies across platforms, because Python calls the platform C library's `strftime()` function, and platform variations are common. To see the full set of format codes supported on your platform, consult the *strftime(3)* documentation.

Pela mesma razão, o tratamento de formatação de strings contendo pontos de código Unicode que não podem ser representados no conjunto de caracteres da localidade atual também é dependente da plataforma. Em algumas plataformas, tais pontos de código são preservados intactos na saída, enquanto em outros `strftime` pode levantar *UnicodeError* ou retornar uma string vazia ao invés.

The following is a list of all the format codes that the C standard (1989 version) requires, and these work on all platforms with a standard C implementation. Note that the 1999 version of the C standard added additional format codes.

[2] Passar `datetime.strptime('Feb 29', '%b %d')` irá falhar como `1900` não é um ano bissexto.

Diretiva	Significado	Exemplo	Notas
%a	Dias da semana como nomes abreviados da localidade.	Sun, Mon, …, Sat (en_US); So, Mo, …, Sa (de_DE)	(1)
%A	Dia da semana como nome completo da localidade.	Sunday, Monday, …, Saturday (en_US); Sonntag, Montag, …, Samstag (de_DE)	(1)
%w	Dia da semana como um número decimal, onde 0 é domingo e 6 é sábado.	0, 1, …, 6	
%d	Dia do mês como um número decimal com zeros a esquerda.	01, 02, …, 31	(9)
%b	Mês como nome da localidade abreviado.	Jan, Feb, …, Dec (en_US); Jan, Feb, …, Dez (de_DE)	(1)
%B	Mês como nome completo da localidade.	January, February, …, December (en_US); janeiro, fevereiro, …, dezembro (pt_BR)	(1)
%m	Mês como um número decimal com zeros a esquerda.	01, 02, …, 12	(9)
%y	Ano sem século como um número decimal com zeros a esquerda.	00, 01, …, 99	(9)
%Y	Ano com o século como um número decimal.	0001, 0002, …, 2013, 2014, …, 9998, 9999	(2)
%H	Hora (relógio de 24 horas) como um número decimal com zeros a esquerda.	00, 01, …, 23	(9)
%I	Hora (relógio de 12 horas) como um número decimal com zeros a esquerda.	01, 02, …, 12	(9)
%p	Equivalente da localidade a AM ou PM.	AM, PM (en_US); am, pm (de_DE)	(1), (3)
%M	Minutos como um número decimal, com zeros a esquerda.	00, 01, …, 59	(9)
%S	Segundos como um número decimal, com zeros a esquerda.	00, 01, …, 59	(4), (9)
%f	Micro-segundos como um número decimal, com ze-	000000, 000001, …, 999999	(5)

Several additional directives not required by the C89 standard are included for convenience. These parameters all correspond to ISO 8601 date values. These may not be available on all platforms when used with the `strftime()` method. The ISO 8601 year and ISO 8601 week directives are not interchangeable with the year and week number directives above. Calling `strptime()` with incomplete or ambiguous ISO 8601 directives will raise a *ValueError*.

Diretiva	Significado	Exemplo	Notas
`%G`	Ano ISO 8601 com o século representando o ano que a maior parte da semana ISO (`%V`).	0001, 0002, …, 2013, 2014, …, 9998, 9999	(8)
`%u`	Dia de semana ISO 8601 como um número decimal onde 1 é segunda-feira.	1, 2, …, 7	
`%V`	Semana ISO 8601 como um número decimal, com segunda-feira como o primeiro dia da semana. A semana 01 é a semana contendo o dia 4 de Janeiro.	01, 02, …, 53	(8), (9)

Novo na versão 3.6: `%G`, `%u` e `%V` foram adicionados.

Notas:

(1) Devido ao fato que o formato depende da localidade atual, deve-se tomar cuidado ao fazer suposições sobre o valor na saída. Ordenamento de campos irá variar (por exemplo, "mês/dia/ano" versus "dia/mês/ano"), e a saída pode conter caracteres Unicode que foram codificados usando a codificação padrão da localidade (por exemplo, se a localidade atual é `ja_JP`, a codificação padrão pode ser qualquer uma dentre `eucJP`, `SJIS`, ou `utf-8`; utilize *locale.getlocale()* para determinar a codificação atual da localidade).

(2) O método `strptime()` pode interpretar anos no intervalo [1, 9999], mas anos < 1000 devem ser preenchidos com 0 para ter 4 dígitos de extensão.

Alterado na versão 3.2: Em versões anteriores, o método `strftime()` era restrito a anos>= 1900.

Alterado na versão 3.3: Na versão 3.2, o método `strftime()` era restrito a anos >= 1000.

(3) Quando usado com o método `strptime()`, a diretiva `%p` apenas afeta as horas na saída se a diretiva `%I` é usada para analisar a hora.

(4) Ao contrário do módulo *time*, o módulo *datetime* não suporta segundos bissextos.

(5) When used with the `strptime()` method, the `%f` directive accepts from one to six digits and zero pads on the right. `%f` is an extension to the set of format characters in the C standard (but implemented separately in datetime objects, and therefore always available).

(6) Para um objeto ingênuo, os códigos de formatação `%z` e `%Z` são substituídos por strings vazias.

Para um objeto consciente:

%z `utcoffset()` is transformed into a string of the form ±HHMM[SS[.ffffff]], where HH is a 2-digit string giving the number of UTC offset hours, MM is a 2-digit string giving the number of UTC offset minutes, SS is a 2-digit string giving the number of UTC offset seconds and ffffff is a 6-digit string giving the number of UTC offset microseconds. The ffffff part is omitted when the offset is a whole number of seconds and both the ffffff and the SS part is omitted when the offset is a whole number of minutes. For example, if `utcoffset()` returns `timedelta(hours=-3, minutes=-30)`, `%z` is replaced with the string `'-0330'`.

Alterado na versão 3.7: A diferença UTC não é restrita a um número completo de minutos.

Alterado na versão 3.7: Quando a diretiva `%z` é fornecida para o método `strptime()`, as diferenças UTC podem ter um separador de vírgula como um separador entre horas, minutos e segundos. Por exemplo, `'+01:00:00'` será interpretado como uma diferença de uma hora. Adicionalmente, fornecer `'Z'` é idêntico a `'+00:00'`.

%Z If `tzname()` returns `None`, `%Z` is replaced by an empty string. Otherwise `%Z` is replaced by the returned value, which must be a string.

Alterado na versão 3.2: When the `%z` directive is provided to the `strptime()` method, an aware *datetime* object will be produced. The `tzinfo` of the result will be set to a *timezone* instance.

(7) Quando for usado com o método `strptime()`, `%U` e `%W` são apenas usados em cálculos quando o dia da semana e o ano do calendário (`%Y`) são especificados.

(8) Similar a `%U` e `%W`, `%V` é apenas usado em cálculos quando o dia da semana e o ano ISO (`%G`) são especificados em uma string de formatação `strptime()`. Perceba também que `%G` e `%Y` não são intercambiáveis.

(9) Quando for usado com o método `strptime()`, o zero precedente é opcional para formatos `%d`, `%m`, `%H`, `%I`, `%M`, `%S`, `%J`, `%U`, `%W`, e `%V`. O formato `%y` não requer um zero precedente.

8.2 `calendar` — General calendar-related functions

Código Fonte: Lib/calendar.py

This module allows you to output calendars like the Unix **cal** program, and provides additional useful functions related to the calendar. By default, these calendars have Monday as the first day of the week, and Sunday as the last (the European convention). Use *setfirstweekday()* to set the first day of the week to Sunday (6) or to any other weekday. Parameters that specify dates are given as integers. For related functionality, see also the *datetime* and *time* modules.

The functions and classes defined in this module use an idealized calendar, the current Gregorian calendar extended indefinitely in both directions. This matches the definition of the "proleptic Gregorian" calendar in Dershowitz and Reingold's book "Calendrical Calculations", where it's the base calendar for all computations. Zero and negative years are interpreted as prescribed by the ISO 8601 standard. Year 0 is 1 BC, year -1 is 2 BC, and so on.

class calendar.**Calendar**(*firstweekday=0*)

Creates a *Calendar* object. *firstweekday* is an integer specifying the first day of the week. `0` is Monday (the default), `6` is Sunday.

A *Calendar* object provides several methods that can be used for preparing the calendar data for formatting. This class doesn't do any formatting itself. This is the job of subclasses.

Calendar instances have the following methods:

iterweekdays()

Return an iterator for the week day numbers that will be used for one week. The first value from the iterator will be the same as the value of the *firstweekday* property.

itermonthdates(*year*, *month*)

Return an iterator for the month *month* (1–12) in the year *year*. This iterator will return all days (as *datetime.date* objects) for the month and all days before the start of the month or after the end of the month that are required to get a complete week.

itermonthdays(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted by the *datetime.date* range. Days returned will simply be day of the month numbers. For the days outside of the specified month, the day number is `0`.

itermonthdays2(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted by the *datetime.date* range. Days returned will be tuples consisting of a day of the month number and a week day number.

itermonthdays3(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted

by the *datetime.date* range. Days returned will be tuples consisting of a year, a month and a day of the month numbers.

Novo na versão 3.7.

itermonthdays4(*year*, *month*)
Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*, but not restricted by the *datetime.date* range. Days returned will be tuples consisting of a year, a month, a day of the month, and a day of the week numbers.

Novo na versão 3.7.

monthdatescalendar(*year*, *month*)
Return a list of the weeks in the month *month* of the *year* as full weeks. Weeks are lists of seven *datetime.date* objects.

monthdays2calendar(*year*, *month*)
Return a list of the weeks in the month *month* of the *year* as full weeks. Weeks are lists of seven tuples of day numbers and weekday numbers.

monthdayscalendar(*year*, *month*)
Return a list of the weeks in the month *month* of the *year* as full weeks. Weeks are lists of seven day numbers.

yeardatescalendar(*year*, *width=3*)
Return the data for the specified year ready for formatting. The return value is a list of month rows. Each month row contains up to *width* months (defaulting to 3). Each month contains between 4 and 6 weeks and each week contains 1–7 days. Days are *datetime.date* objects.

yeardays2calendar(*year*, *width=3*)
Return the data for the specified year ready for formatting (similar to *yeardatescalendar()*). Entries in the week lists are tuples of day numbers and weekday numbers. Day numbers outside this month are zero.

yeardayscalendar(*year*, *width=3*)
Return the data for the specified year ready for formatting (similar to *yeardatescalendar()*). Entries in the week lists are day numbers. Day numbers outside this month are zero.

class calendar.**TextCalendar**(*firstweekday=0*)
This class can be used to generate plain text calendars.

TextCalendar instances have the following methods:

formatmonth(*theyear*, *themonth*, *w=0*, *l=0*)
Return a month's calendar in a multi-line string. If *w* is provided, it specifies the width of the date columns, which are centered. If *l* is given, it specifies the number of lines that each week will use. Depends on the first weekday as specified in the constructor or set by the *setfirstweekday()* method.

prmonth(*theyear*, *themonth*, *w=0*, *l=0*)
Print a month's calendar as returned by *formatmonth()*.

formatyear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)
Return a *m*-column calendar for an entire year as a multi-line string. Optional parameters *w*, *l*, and *c* are for date column width, lines per week, and number of spaces between month columns, respectively. Depends on the first weekday as specified in the constructor or set by the *setfirstweekday()* method. The earliest year for which a calendar can be generated is platform-dependent.

pryear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)
Print the calendar for an entire year as returned by *formatyear()*.

class calendar.**HTMLCalendar**(*firstweekday=0*)
This class can be used to generate HTML calendars.

HTMLCalendar instances have the following methods:

formatmonth(*theyear*, *themonth*, *withyear=True*)
Return a month's calendar as an HTML table. If *withyear* is true the year will be included in the header, otherwise just the month name will be used.

formatyear(*theyear*, *width=3*)
Return a year's calendar as an HTML table. *width* (defaulting to 3) specifies the number of months per row.

formatyearpage(*theyear*, *width=3*, *css='calendar.css'*, *encoding=None*)
Return a year's calendar as a complete HTML page. *width* (defaulting to 3) specifies the number of months per row. *css* is the name for the cascading style sheet to be used. `None` can be passed if no style sheet should be used. *encoding* specifies the encoding to be used for the output (defaulting to the system default encoding).

`HTMLCalendar` has the following attributes you can override to customize the CSS classes used by the calendar:

cssclasses
A list of CSS classes used for each weekday. The default class list is:

```
cssclasses = ["mon", "tue", "wed", "thu", "fri", "sat", "sun"]
```

more styles can be added for each day:

```
cssclasses = ["mon text-bold", "tue", "wed", "thu", "fri", "sat", "sun red"]
```

Note that the length of this list must be seven items.

cssclass_noday
The CSS class for a weekday occurring in the previous or coming month.

Novo na versão 3.7.

cssclasses_weekday_head
A list of CSS classes used for weekday names in the header row. The default is the same as *cssclasses*.

Novo na versão 3.7.

cssclass_month_head
The month's head CSS class (used by `formatmonthname()`). The default value is `"month"`.

Novo na versão 3.7.

cssclass_month
The CSS class for the whole month's table (used by *formatmonth()*). The default value is `"month"`.

Novo na versão 3.7.

cssclass_year
The CSS class for the whole year's table of tables (used by *formatyear()*). The default value is `"year"`.

Novo na versão 3.7.

cssclass_year_head
The CSS class for the table head for the whole year (used by *formatyear()*). The default value is `"year"`.

Novo na versão 3.7.

Note that although the naming for the above described class attributes is singular (e.g. `cssclass_month` `cssclass_noday`), one can replace the single CSS class with a space separated list of CSS classes, for example:

```
"text-bold text-red"
```

Here is an example how `HTMLCalendar` can be customized:

```
class CustomHTMLCal(calendar.HTMLCalendar):
    cssclasses = [style + " text-nowrap" for style in
                  calendar.HTMLCalendar.cssclasses]
    cssclass_month_head = "text-center month-head"
    cssclass_month = "text-center month"
    cssclass_year = "text-italic lead"
```

class calendar.**LocaleTextCalendar**(*firstweekday=0*, *locale=None*)
This subclass of *TextCalendar* can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.

class calendar.**LocaleHTMLCalendar**(*firstweekday=0*, *locale=None*)
This subclass of *HTMLCalendar* can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.

Nota: The `formatweekday()` and `formatmonthname()` methods of these two classes temporarily change the current locale to the given *locale*. Because the current locale is a process-wide setting, they are not thread-safe.

Para simples calendários de texto, este módulo fornece as seguintes funções.

calendar.**setfirstweekday**(*weekday*)
Sets the weekday (`0` is Monday, `6` is Sunday) to start each week. The values `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, and `SUNDAY` are provided for convenience. For example, to set the first weekday to Sunday:

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

calendar.**firstweekday**()
Returns the current setting for the weekday to start each week.

calendar.**isleap**(*year*)
Returns *True* if *year* is a leap year, otherwise *False*.

calendar.**leapdays**(*y1*, *y2*)
Returns the number of leap years in the range from *y1* to *y2* (exclusive), where *y1* and *y2* are years.

This function works for ranges spanning a century change.

calendar.**weekday**(*year*, *month*, *day*)
Returns the day of the week (`0` is Monday) for *year* (`1970`–…), *month* (`1`–`12`), *day* (`1`–`31`).

calendar.**weekheader**(*n*)
Return a header containing abbreviated weekday names. *n* specifies the width in characters for one weekday.

calendar.**monthrange**(*year*, *month*)
Returns weekday of first day of the month and number of days in month, for the specified *year* and *month*.

calendar.**monthcalendar**(*year*, *month*)
Returns a matrix representing a month's calendar. Each row represents a week; days outside of the month a represented by zeros. Each week begins with Monday unless set by *setfirstweekday()*.

calendar.**prmonth**(*theyear*, *themonth*, *w=0*, *l=0*)
Prints a month's calendar as returned by *month()*.

calendar.**month**(*theyear*, *themonth*, *w=0*, *l=0*)
Returns a month's calendar in a multi-line string using the `formatmonth()` of the *TextCalendar* class.

calendar.**prcal**(*year*, *w=0*, *l=0*, *c=6*, *m=3*)
Prints the calendar for an entire year as returned by `calendar()`.

calendar.**calendar**(*year*, *w=2*, *l=1*, *c=6*, *m=3*)
Returns a 3-column calendar for an entire year as a multi-line string using the `formatyear()` of the `TextCalendar` class.

calendar.**timegm**(*tuple*)
An unrelated but handy function that takes a time tuple such as returned by the `gmtime()` function in the `time` module, and returns the corresponding Unix timestamp value, assuming an epoch of 1970, and the POSIX encoding. In fact, `time.gmtime()` and `timegm()` are each others' inverse.

The `calendar` module exports the following data attributes:

calendar.**day_name**
An array that represents the days of the week in the current locale.

calendar.**day_abbr**
An array that represents the abbreviated days of the week in the current locale.

calendar.**month_name**
An array that represents the months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and `month_name[0]` is the empty string.

calendar.**month_abbr**
An array that represents the abbreviated months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and `month_abbr[0]` is the empty string.

Ver também:

Módulo *datetime* Object-oriented interface to dates and times with similar functionality to the `time` module.

Módulo *time* Low-level time related functions.

8.3 Tipos de dados do contêiner

Source code: Lib/collections/__init__.py

Este módulo implementa tipos de dados de contêiner especializados que fornecem alternativas aos contêineres integrados de uso geral do Python, `dict`, `list`, `set`, and `tuple`.

`namedtuple()`	Função de fábrica para criar subclasses de tuplas com campos nomeados
`deque`	Contêiner semelhante a list com rápido appends e pops em qualquer fim
`ChainMap`	Classe semelhante ao dict(dicionário) para criar uma visão única de vários mapeamentos
`Counter`	Subclasse de Dict para contar objetos hashable
`OrderedDict`	Subclasse de Dict que lembra a ordem que as entradas foram adicionadas
`defaultdict`	Subclasse de Dict que chama uma função de fábrica (factory function) para fornecer valores em faltam
`UserDict`	Envoltório em torno de objetos de dictionary para uma subclasse de dict mais fácil
`UserList`	Invólucro em torno de objetos de list para uma subclasse de list mais fácil
`UserString`	Invólucro em torno de objetos strings para uma subclasse mais fácil

Deprecated since version 3.3, will be removed in version 3.9: Moved *Coleções Abstratas Classes Base* to the `collections.abc` module. For backwards compatibility, they continue to be visible in this module through Python 3.8.

8.3.1 `ChainMap` objects

Novo na versão 3.3.

Uma classe *`ChainMap`* é fornecido para ligar rapidamente uma série de mapeamentos para que eles possam ser tratados como uma única unidade. Muitas vezes é muito mais rápido do que criar um novo dicionário e executar múltiplas chamadas *`update()`*

A classe pode ser usada para simular escopos aninhados e é útil em modelos.

class collections.**ChainMap**(**maps*)

Um grupo de múltiplos dicts *`ChainMap`* ou outros mapeamentos juntos para criar uma única view atualizável. Se *maps* não são especificados, Um diticionário vazio é fornecido para que uma nova cadeia sempre tenha pelo menos um mapeamento.

Os mapeamentos subjacentes são armazenados em uma lista. Essa lista é pública e pode ser acessada ou atualizada usando o atributo * maps *. Não existe outro estado.

Faz a busca nos mapeamentos subjacentes sucessivamente até que uma chave seja encontrada. Em contraste, escrita, atualições e remoções operam apenas no primeiro mapeamento.

Uma *`ChainMap`* incorpora os mapeamentos subjacentes por referência. Então, se um dos mapeamentos subjacentes for atualizado, essas alterações serão refletidas em: class: *ChainMap*.

Todos os métodos usuais do dicionário são suportados. Além disso, existe um atributo *maps*, um método para criar novos subcontextos e uma propriedade para acessar todos, exceto o primeiro mapeamento:

maps

Uma lista de mapeamentos atualizáveis pelo usuário. A lista é ordenada desde o primeiro pesquisado até a última pesquisado. É o único estado armazenado e pode ser modificado para alterar quais mapeamentos são pesquisados. A lista deve sempre conter pelo menos um mapeamento.

new_child(*m=None*)

Retorna uma nova: class: 'ChainMap' contendo um novo mapa seguido de todos os mapas na instância atual. Se `m` for especificado, torna-se o novo mapa na frente da lista de mapeamentos; Se não especificado, é usado um dicionário vazio, de modo que chamar `d.new_child ()` é equivalente a: `ChainMap({}, *d.maps)`. Esse método é usado para criar subcontextos que podem ser atualizados sem alterar valores em nenhum dos mapeamentos pai.

Alterado na versão 3.4: O parâmetro opcional "m" foi adicionado.

parents

<nested scope>

Observe, a ordem de iteração de um *`ChainMap()`* é determinada pela varredura dos mapeamentos do último ao primeiro:

```
>>> baseline = {'music': 'bach', 'art': 'rembrandt'}
>>> adjustments = {'art': 'van gogh', 'opera': 'carmen'}
>>> list(ChainMap(adjustments, baseline))
['music', 'art', 'opera']
```

Isso dá a mesma ordem de uma série de chamadas *`dict.update()`* começando com o último mapeamento:

```
>>> combined = baseline.copy()
>>> combined.update(adjustments)
>>> list(combined)
['music', 'art', 'opera']
```

Ver também:

- A classe MultiContext no pacote CodeTools de Enthought tem opções para suportar a escrita em qualquer mapeamento na cadeia.
- Django's Context class for templating is a read-only chain of mappings. It also features pushing and popping of contexts similar to the *new_child()* method and the *parents* property.
- A receita de Contextos Aninhados possui opções para controlar se escritas e outras mutações se aplicam a apenas o primeiro mapeamento ou para qualquer mapeamento na cadeia.
- Uma versão muito simplificada somente leitura do Chainmap.<https://code.activestate.com/recipes/305268/>'_.

Exemplos e Receitas de `ChainMap`

Esta seção mostra várias abordagens para trabalhar com mapas encadeados.

Exemplo de simulação da cadeia de busca interna do Python:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

Exemplo de como permitir que os argumentos de linha de comando especificados pelo usuário tenham precedência sobre as variáveis de ambiente que, por sua vez, têm precedência sobre os valores padrão:

```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k: v for k, v in vars(namespace).items() if v is not None}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

Padrões de exemplo para utilização da classe *ChainMap* para simular contextos aninhados:

```
c = ChainMap()          # Create root context
d = c.new_child()       # Create nested child context
e = c.new_child()       # Child of c, independent from d
e.maps[0]               # Current context dictionary -- like Python's locals()
e.maps[-1]              # Root context -- like Python's globals()
e.parents               # Enclosing context chain -- like Python's nonlocals

d['x'] = 1              # Set value in current context
d['x']                  # Get first key in the chain of contexts
del d['x']              # Delete from current context
list(d)                 # All nested values
k in d                  # Check all nested values
len(d)                  # Number of nested values
d.items()               # All nested items
dict(d)                 # Flatten into a regular dictionary
```

A classe *ChainMap* só faz atualizações (escritas e remoções) no primeiro mapeamento na cadeia, enquanto as pesquisas irão buscar em toda a cadeia. Contudo, se há o desejo de escritas e remoções profundas, é fácil fazer uma subclasse que atualiza chaves encontradas mais a fundo na cadeia:

```
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'

    def __setitem__(self, key, value):
        for mapping in self.maps:
            if key in mapping:
                mapping[key] = value
                return
        self.maps[0][key] = value

    def __delitem__(self, key):
        for mapping in self.maps:
            if key in mapping:
                del mapping[key]
                return
        raise KeyError(key)

>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'}, {'lion': 'yellow'})
>>> d['lion'] = 'orange'         # update an existing key two levels down
>>> d['snake'] = 'red'           # new keys get added to the topmost dict
>>> del d['elephant']            # remove an existing key one level down
>>> d                            # display result
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange'})
```

8.3.2 Objetos `Counter`

Uma ferramenta de contagem é fornecida para apoiar contas rápidas e convenientes. Por exemplo:

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

class collections.**Counter**([*iterable-or-mapping*])

Um *Counter* é uma subclasse de *dict* subclass para contagem de objetos hasheáveis. É uma coleção na qual elementos são armazenados como chaves de dicionário e suas contagens são armazenadas como valores de dicionário. Contagens podem ser qualquer valor inteiro incluindo zero e contagens negativas. A classe *Counter* é similar a sacos ou multiconjuntos em outras linguagens.

Os elementas são contados a partir de um *iterável* ou inicializado a partir de um outro *mapeamento* (ou contador):

```
>>> c = Counter()                           # a new, empty counter
>>> c = Counter('gallahad')                 # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)             # a new counter from keyword args
```

Objetos Counter tem uma interface de dicionário, com a diferença que devolvem uma contagem zero para itens

que não estão presentes em vez de levantar a excessão *KeyError*:

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                                # count of a missing element is zero
0
```

Definir uma contagem como zero não remove um elemento do contador. Use `del` para o remover completamente.

```
>>> c['sausage'] = 0                          # counter entry with a zero count
>>> del c['sausage']                          # del actually removes the entry
```

Novo na versão 3.1.

Objetos Counter permitem três métodos além dos disponíveis para todos os dicionário:

elements()
Return an iterator over elements repeating each as many times as its count. Elements are returned in arbitrary order. If an element's count is less than one, *elements()* will ignore it.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common([*n*])
Return a list of the *n* most common elements and their counts from the most common to the least. If *n* is omitted or `None`, *most_common()* returns *all* elements in the counter. Elements with equal counts are ordered arbitrarily:

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('r', 2), ('b', 2)]
```

subtract([*iterable-or-mapping*])
Os elementos são subtraídos de um *iterável* ou de outro *mapeamento* (ou contador). Funciona como *dict.update()*, mas subtraindo contagens ao invés de substituí-las. Tanto as entradas quanto as saídas podem ser zero ou negativas.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

Novo na versão 3.2.

Os métodos usuais de dicionário estão disponíveis para objetos *Counter*, exceto por dois que funcionam de forma diferente para contadores.

fromkeys(*iterable*)
Este método de classe não está implementado para objetos *Counter*.

update([*iterable-or-mapping*])
Elementos são contados a partir de um *iterável* ou adicionados de outro *mapeamento* (ou contador). Funciona como *dict.update()* mas adiciona contagens em vez de substituí-las. Além disso, é esperado que o *iterável* seja uma sequência de elementos, e não uma sequência de pares `(key, value)`.

Padrões comuns para trabalhar com objetos *Counter*:

```
sum(c.values())                 # total of all counts
c.clear()                       # reset all counts
```

(continua na próxima página)

(continuação da página anterior)

```
list(c)                          # list unique elements
set(c)                           # convert to a set
dict(c)                          # convert to a regular dictionary
c.items()                        # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs))     # convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1]        # n least common elements
+c                               # remove zero and negative counts
```

Várias operações matemáticas são fornecidas para combinar: class: objetos *Counter* para produzir multisets (counters que têm contagens maiores que zero). A adição e a subtração combinam counters adicionando ou subtraindo as contagens dos elementos correspondentes. A intersecção e a união retornam o mínimo e o máximo das contagens correspondentes. Cada operação pode aceitar entradas com contagens assinadas, mas a saída excluirá resultados com contagens de zero ou menos.

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                        # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                        # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d                        # intersection:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                        # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
```

A adição e subtração unárias são atalhos para adicionar um contador vazio ou subtrair de um contador vazio.

```
>>> c = Counter(a=2, b=-4)
>>> +c
Counter({'a': 2})
>>> -c
Counter({'b': 4})
```

Novo na versão 3.3: Adicionado suporte para operador unário mais, unário menos e operação in-place multiset.

Nota: Os contadores foram projetados principalmente para funcionar com números inteiros positivos para representar contagens contínuas; no entanto, foi tomado cuidado para não impedir desnecessariamente os casos de uso que precisassem de outros tipos ou valores negativos. Para ajudar nesses casos de uso, esta seção documenta o intervalo mínimo e as restrições de tipo.

- A própria classe `Counter` é uma subclasse de dicionário sem restrições em suas chaves e valores. Os valores pretendem ser números que representam contagens, mas você *pode* armazenar qualquer coisa no campo de valor.
- O método `most_common()` requer apenas que os valores sejam ordenáveis.
- Para operações in-place, como `c[key] += 1`, o tipo de valor precisa suportar apenas adição e subtração. Portanto, frações, números de ponto flutuante e decimais funcionariam e os valores negativos são suportados. O mesmo também é verdadeiro para: meth: ~ *Counter.update* e: meth:‘ ~ Counter.subtract‘ que permitem valores negativos e zero para entradas e saídas.
- Os métodos multiset são projetados apenas para casos de uso com valores positivos. As entradas podem ser negativas ou zero, mas apenas saídas com valores positivos são criadas. Não há restrições de tipo, mas o tipo de valor precisa suportar adição, subtração e comparação.
- O método: meth: ~ *Counter.elements* requer contagens inteiras. Ele ignora contagens zero e negativas.

Ver também:

- Classe Bag <https://www.gnu.org/software/smalltalk/manual-base/html_node/Bag.html> '_ em Smalltalk.
- Entrada da Wikipedia para *Multisets <https://en.wikipedia.org/wiki/Multiset>* _.
- Tutorial com exemplos C++ multisets.
- Para operações matemáticas em multisets e seus casos de uso, consulte * Knuth, Donald. The Art of Computer Programming Volume II, Seção 4.6.3, Exercício 19 *.
- Para enumerar todos os multisets distintos de um determinado tamanho em um determinado conjunto de elementos, consulte: func:' itertools.combinations_with_replacement'

```
map(Counter, combinations_with_replacement('ABC', 2)) # --> AA AB AC BB BC CC
```

8.3.3 Objetos deque

class collections.**deque**([*iterable*[, *maxlen*]])

Retorna um novo objeto deque inicializado da esquerda para a direita (usando: meth: *append*) com dados de * iterável *. Se * iterável * não for especificado, o novo deque estará vazio.

Deques são uma generalização de pilhas e filas (o nome é pronunciado "deck" e é abreviação de "fila dupla"). O Deques oferece suporte para acréscimos e saliências com segurança de thread e com eficiência de memória de ambos os lados do deque com aproximadamente o mesmo desempenho O (1) em qualquer direção.

Embora os objetos: class: *list* suportem operações semelhantes, eles são otimizados para operações rápidas de comprimento fixo e sujeitam em custos de movimentação de memória O (n) para' *pop (0) e insert (0, v)* ' operações que alteram o tamanho e a posição da representação de dados subjacente.

Se * maxlen * não for especificado ou for " Nenhum", deques podem crescer para um comprimento arbitrário. Caso contrário, o deque é limitado ao comprimento máximo especificado. Quando um deque de comprimento limitado está cheio, quando novos itens são adicionados, um número correspondente de itens é descartado da extremidade oposta. Deques de comprimento limitado fornecem funcionalidade semelhante ao filtro " tail" no Unix. Eles também são úteis para rastrear transações e outros pools de dados onde apenas a atividade mais recente é de interesse.

Os objetos Deque suportam os seguintes métodos:

append(*x*)
Adicione * x * ao lado direito do deque.

appendleft(*x*)
Adicione * x * ao lado esquerdo do deque

clear()
Remova todos os elementos do deque deixando-o com comprimento 0.

copy()
Cria uma cópia rasa do deque.

Novo na versão 3.5.

count(*x*)
Conta o número de elementos deque igual a * x *.

Novo na versão 3.2.

extend(*iterable*)
Estenda o lado direito do deque anexando elementos do argumento iterável.

extendleft(*iterable*)
Estenda o lado esquerdo do deque anexando elementos de *iterable*. Observe que a série de acréscimos à esquerda resulta na reversão da ordem dos elementos no argumento iterável.

index(*x*[, *start*[, *stop*]])
Retorne a posição de * x * no deque (no ou após o índice * início * e antes do índice * parada *). Retorna a primeira correspondência ou aumenta: exc: *ValueError* se não for encontrado.

Novo na versão 3.5.

insert(*i*, *x*)
Insira * x * no deque na posição * i *.

Se a inserção fizer com que um deque limitado cresça além de * maxlen *, um: exc: *IndexError* é gerado.

Novo na versão 3.5.

pop()
Remova e devolva um elemento do lado direito do deque. Se nenhum elemento estiver presente, levanta um: exc: *IndexError*.

popleft()
Remova e devolva um elemento do lado esquerdo do deque. Se nenhum elemento estiver presente, levanta um: exc: *IndexError*.

remove(*value*)
Remova a primeira ocorrência de * valor *. Se não for encontrado, levanta um: exc: *ValueError*.

reverse()
Inverta os elementos do deque no local e, em seguida, retorne `None`.

Novo na versão 3.2.

rotate(*n=1*)
Gire os passos deque * n * para a direita. Se * n * for negativo, gire para a esquerda.

Quando o deque não está vazio, girar um passo para a direita é equivalente a `d.appendleft(d.pop())`, e girar um passo para a esquerda é equivalente a `d.append(d.popleft())`.

Os objetos Deque também fornecem um atributo somente leitura:

maxlen
Tamanho máximo de um deque ou `None` se ilimitado.

Novo na versão 3.1.

In addition to the above, deques support iteration, pickling, `len(d)`, `reversed(d)`, `copy.copy(d)`, `copy.deepcopy(d)`, membership testing with the `in` operator, and subscript references such as `d[-1]`. Indexed access is O(1) at both ends but slows to O(n) in the middle. For fast random access, use lists instead.

A partir da versão 3.5, deques suporta “ __add __ () “, “ __mul __ () “ e “ __imul __ () “.

Exemplo:

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print(elem.upper())
G
H
I

>>> d.append('j')                    # add a new entry to the right side
```

(continua na próxima página)

(continuação da página anterior)

```
>>> d.appendleft('f')                 # add a new entry to the left side
>>> d                                 # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                           # return and remove the rightmost item
'j'
>>> d.popleft()                       # return and remove the leftmost item
'f'
>>> list(d)                           # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                              # peek at leftmost item
'g'
>>> d[-1]                             # peek at rightmost item
'i'

>>> list(reversed(d))                 # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                          # search the deque
True
>>> d.extend('jkl')                   # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                       # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                      # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))                # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                         # empty the deque
>>> d.pop()                           # cannot pop from an empty deque
Traceback (most recent call last):
    File "<pyshell#6>", line 1, in -toplevel-
        d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')               # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

Receitas de `deque`

Esta seção mostra várias abordagens para trabalhar com deques.

Deques de comprimento limitado fornecem funcionalidade semelhante ao filtro `tail` em Unix

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

Outra abordagem para usar deques é manter uma sequência de elementos adicionados recentemente, acrescentando à direita e clicando à esquerda:

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # http://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

Um *escalonador round-robin <https://en.wikipedia.org/wiki/Round-robin_scheduling> _* pode ser implementado com iteradores de entrada armazenados em um: class: *deque*. Os valores são gerados a partir do iterador ativo na posição zero. Se esse iterador estiver esgotado, ele pode ser removido com: meth: ~ *deque.popleft*; caso contrário, ele pode voltar ao fim com o método: meth: ~ *deque.rotate*

```
def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    iterators = deque(map(iter, iterables))
    while iterators:
        try:
            while True:
                yield next(iterators[0])
                iterators.rotate(-1)
        except StopIteration:
            # Remove an exhausted iterator.
            iterators.popleft()
```

O método: meth: ~ *deque.rotate* fornece uma maneira de implementar o corte e exclusão: class:‘ deque‘. Por exemplo, uma implementação Python pura de `del d[n]` depende do método `rotate()` para posicionar os elementos a serem popped

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

Para implementar o corte: class: *deque*, use uma abordagem semelhante aplicando: meth:‘ ~ deque.rotate‘ para trazer um elemento alvo para o lado esquerdo do deque. Remova as entradas antigas com: meth: ~ *deque.popleft*, adicione novas entradas com: meth:‘ ~ deque.extend‘, e então inverta a rotação. Com pequenas variações dessa abordagem, é fácil implementar manipulações de pilha de estilo Forth, como `dup`,“drop“,“swap“,“over“,“pick“,“rot“, and `roll`.

8.3.4 Objetos `defaultdict`

class collections.**defaultdict**([*default_factory*[, ...]])

Returns a new dictionary-like object. *defaultdict* is a subclass of the built-in *dict* class. It overrides one method and adds one writable instance variable. The remaining functionality is the same as for the *dict* class and is not documented here.

O primeiro argumento fornece o valor inicial para o atributo: attr: *default_factory*; o padrão é “ Nenhum“. Todos os argumentos restantes são tratados da mesma forma como se fossem passados para o construtor: class: *dict*, incluindo argumentos de palavra-chave.

Os objetos: class: *defaultdict* suportam o seguinte método além das operações padrão: class:‘ dict‘:

__missing__(*key*)
Se o atributo: attr: *default_factory* é `None`, isso levanta uma exceção: exc:‘ KeyError‘ com a * chave * como argumento.

Se: attr: *default_factory* não for `None`, ele é chamado sem argumentos para fornecer um valor padrão para a * chave * fornecida, este valor é inserido no dicionário para a * chave * e retornado.

Se chamar: attr: *default_factory* levanta uma exceção, esta exceção é propagada inalterada.

Este método é chamado pelo método: meth: *__getitem__* da classe: class:‘ dict‘ quando a chave solicitada não é encontrada; tudo o que ele retorna ou aumenta é então retornado ou gerado por: meth: *__getitem__*.

Note that *__missing__()* is *not* called for any operations besides `__getitem__()`. This means that `get()` will, like normal dictionaries, return `None` as a default rather than using *default_factory*.

Objetos *defaultdict* permitem a seguinte variável instanciada:

default_factory
This attribute is used by the *__missing__()* method; it is initialized from the first argument to the constructor, if present, or to `None`, if absent.

Exemplos de `defaultdict`

Usando *list* como *default_factory*, se for fácil agrupar a sequencia dos pares chave-valores num dicionário de listas

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

When each key is encountered for the first time, it is not already in the mapping; so an entry is automatically created using the *default_factory* function which returns an empty *list*. The `list.append()` operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that key) and the `list.append()` operation adds another value to the list. This technique is simpler and faster than an equivalent technique using *dict.setdefault()*:

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Setting the *default_factory* to *int* makes the *defaultdict* useful for counting (like a bag or multiset in other languages):

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

When a letter is first encountered, it is missing from the mapping, so the *default_factory* function calls *int()* to supply a default count of zero. The increment operation then builds up the count for each letter.

The function *int()* which always returns zero is just a special case of constant functions. A faster and more flexible way to create constant functions is to use a lambda function which can supply any constant value (not just zero):

```
>>> def constant_factory(value):
...     return lambda: value
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

Setting the *default_factory* to *set* makes the *defaultdict* useful for building a dictionary of sets:

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```

8.3.5 `namedtuple()` Factory Function for Tuples with Named Fields

Tuplas nomeadas determinam o significado de cada posição numa tupla e permitem um código mais legível e autodocumentado. Podem ser usadas sempre que tuplas regulares forem utilizadas, e adicionam a possibilidade de acessar campos pelo nome ao invés da posição do índice.

`collections.`**`namedtuple`**(*typename*, *field_names*, *, *rename=False*, *defaults=None*, *module=None*)

Returns a new tuple subclass named *typename*. The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being indexable and iterable. Instances of the subclass also have a helpful docstring (with typename and field_names) and a helpful `__repr__()` method which lists the tuple contents in a `name=value` format.

The *field_names* are a sequence of strings such as `['x', 'y']`. Alternatively, *field_names* can be a single string with each fieldname separated by whitespace and/or commas, for example `'x y'` or `'x, y'`.

Any valid Python identifier may be used for a fieldname except for names starting with an underscore. Valid identifiers consist of letters, digits, and underscores but do not start with a digit or underscore and cannot be a *keyword* such as *class*, *for*, *return*, *global*, *pass*, or *raise*.

If *rename* is true, invalid fieldnames are automatically replaced with positional names. For example, `['abc', 'def', 'ghi', 'abc']` is converted to `['abc', '_1', 'ghi', '_3']`, eliminating the keyword `def` and the duplicate fieldname `abc`.

defaults can be `None` or an *iterable* of default values. Since fields with a default value must come after any fields without a default, the *defaults* are applied to the rightmost parameters. For example, if the fieldnames are `['x', 'y', 'z']` and the defaults are `(1, 2)`, then `x` will be a required argument, `y` will default to `1`, and `z` will default to `2`.

If *module* is defined, the `__module__` attribute of the named tuple is set to that value.

Named tuple instances do not have per-instance dictionaries, so they are lightweight and require no more memory than regular tuples.

Alterado na versão 3.1: Adicionado suporte a *rename*.

Alterado na versão 3.6: The *verbose* and *rename* parameters became *keyword-only arguments*.

Alterado na versão 3.6: Adicionado o parametro *module*

Alterado na versão 3.7: Remove the *verbose* parameter and the `_source` attribute.

Alterado na versão 3.7: Adicionado o parâmetro *defaults* e o atributo `_field_defaults`.

```
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)     # instantiate with positional or keyword arguments
>>> p[0] + p[1]             # indexable like the plain tuple (11, 22)
33
>>> x, y = p                # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y               # fields also accessible by name
33
>>> p                       # readable __repr__ with a name=value style
Point(x=11, y=22)
```

Named tuples are especially useful for assigning field names to result tuples returned by the *csv* or *sqlite3* modules:

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade
↪')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print(emp.name, emp.title)

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print(emp.name, emp.title)
```

In addition to the methods inherited from tuples, named tuples support three additional methods and two attributes. To prevent conflicts with field names, the method and attribute names start with an underscore.

classmethod somenamedtuple.**_make**(*iterable*)

Class method that makes a new instance from an existing sequence or iterable.

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

somenamedtuple.**_asdict**()

Retorna um novo *dict* que mapeia nomes de campo para seus respectivos valores:

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
OrderedDict([('x', 11), ('y', 22)])
```

Alterado na versão 3.1: Returns an *OrderedDict* instead of a regular *dict*.

somenamedtuple.**_replace**(***kwargs*)

Return a new instance of the named tuple replacing specified fields with new values:

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
```

(continua na próxima página)

(continuação da página anterior)

```
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=newprices[partnum],␣
↪timestamp=time.now())
```

somenamedtuple.**_fields**

Tuple of strings listing the field names. Useful for introspection and for creating new named tuple types from existing named tuples.

```
>>> p._fields            # view the field names
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

somenamedtuple.**_field_defaults**

Dictionary mapping field names to default values.

```
>>> Account = namedtuple('Account', ['type', 'balance'], defaults=[0])
>>> Account._field_defaults
{'balance': 0}
>>> Account('premium')
Account(type='premium', balance=0)
```

To retrieve a field whose name is stored in a string, use the *getattr()* function:

```
>>> getattr(p, 'x')
11
```

To convert a dictionary to a named tuple, use the ** operator (as described in tut-unpacking-arguments):

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

Since a named tuple is a regular Python class, it is easy to add or change functionality with a subclass. Here is how to add a calculated field and a fixed-width print format:

```
>>> class Point(namedtuple('Point', ['x', 'y'])):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f' % (self.x, self.y, self.
↪hypot)

>>> for p in Point(3, 4), Point(14, 5/7):
...     print(p)
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

The subclass shown above sets `__slots__` to an empty tuple. This helps keep memory requirements low by preventing the creation of instance dictionaries.

Subclassing is not useful for adding new, stored fields. Instead, simply create a new named tuple type from the *_fields* attribute:

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

Docstrings can be customized by making direct assignments to the __doc__ fields:

```
>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
>>> Book.id.__doc__ = '13-digit ISBN'
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last name'
```

Alterado na versão 3.5: Property docstrings became writeable.

Default values can be implemented by using *_replace()* to customize a prototype instance:

```
>>> Account = namedtuple('Account', 'owner balance transaction_count')
>>> default_account = Account('<owner name>', 0.0, 0)
>>> johns_account = default_account._replace(owner='John')
>>> janes_account = default_account._replace(owner='Jane')
```

Ver também:

- See *typing.NamedTuple* for a way to add type hints for named tuples. It also provides an elegant notation using the `class` keyword:

  ```
  class Component(NamedTuple):
      part_number: int
      weight: float
      description: Optional[str] = None
  ```

- See *types.SimpleNamespace()* for a mutable namespace based on an underlying dictionary instead of a tuple.
- The *dataclasses* module provides a decorator and functions for automatically adding generated special methods to user-defined classes.

8.3.6 Objetos `OrderedDict`

Ordered dictionaries are just like regular dictionaries but have some extra capabilities relating to ordering operations. They have become less important now that the built-in *dict* class gained the ability to remember insertion order (this new behavior became guaranteed in Python 3.7).

Some differences from *dict* still remain:

- The regular *dict* was designed to be very good at mapping operations. Tracking insertion order was secondary.
- The *OrderedDict* was designed to be good at reordering operations. Space efficiency, iteration speed, and the performance of update operations were secondary.
- Algorithmically, *OrderedDict* can handle frequent reordering operations better than *dict*. This makes it suitable for tracking recent accesses (for example in an LRU cache).
- The equality operation for *OrderedDict* checks for matching order.
- The `popitem()` method of *OrderedDict* has a different signature. It accepts an optional argument to specify which item is popped.
- *OrderedDict* has a `move_to_end()` method to efficiently reposition an element to an endpoint.

- Until Python 3.8, *dict* lacked a `__reversed__()` method.

class collections.**OrderedDict**([*items*])

Return an instance of a *dict* subclass that has methods specialized for rearranging dictionary order.

Novo na versão 3.1.

popitem(*last=True*)

The *popitem()* method for ordered dictionaries returns and removes a (key, value) pair. The pairs are returned in LIFO order if *last* is true or FIFO (first-in, first-out) order if false.

move_to_end(*key*, *last=True*)

Move an existing *key* to either end of an ordered dictionary. The item is moved to the right end if *last* is true (the default) or to the beginning if *last* is false. Raises *KeyError* if the *key* does not exist:

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d.keys())
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d.keys())
'bacde'
```

Novo na versão 3.2.

Além dos métodos usuais de mapeamento, dicionários ordenados também suportam iteração reversa usando a função *reversed()*.

Equality tests between *OrderedDict* objects are order-sensitive and are implemented as `list(od1.items())==list(od2.items())`. Equality tests between *OrderedDict* objects and other *Mapping* objects are order-insensitive like regular dictionaries. This allows *OrderedDict* objects to be substituted anywhere a regular dictionary is used.

Alterado na versão 3.5: The items, keys, and values *views* of *OrderedDict* now support reverse iteration using *reversed()*.

Alterado na versão 3.6: With the acceptance of **PEP 468**, order is retained for keyword arguments passed to the *OrderedDict* constructor and its `update()` method.

`OrderedDict` Examples and Recipes

It is straightforward to create an ordered dictionary variant that remembers the order the keys were *last* inserted. If a new entry overwrites an existing entry, the original insertion position is changed and moved to the end:

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        super().move_to_end(key)
```

An *OrderedDict* would also be useful for implementing variants of *functools.lru_cache()*:

```
class LRU(OrderedDict):
    'Limit size, evicting the least recently looked-up key when full'

    def __init__(self, maxsize=128, *args, **kwds):
        self.maxsize = maxsize
```

(continua na próxima página)

(continuação da página anterior)

```
        super().__init__(*args, **kwds)

    def __getitem__(self, key):
        value = super().__getitem__(key)
        self.move_to_end(key)
        return value

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        if len(self) > self.maxsize:
            oldest = next(iter(self))
            del self[oldest]
```

8.3.7 `UserDict` objects

The class, *UserDict* acts as a wrapper around dictionary objects. The need for this class has been partially supplanted by the ability to subclass directly from *dict*; however, this class can be easier to work with because the underlying dictionary is accessible as an attribute.

class collections.**UserDict**([*initialdata*])

Class that simulates a dictionary. The instance's contents are kept in a regular dictionary, which is accessible via the *data* attribute of *UserDict* instances. If *initialdata* is provided, *data* is initialized with its contents; note that a reference to *initialdata* will not be kept, allowing it be used for other purposes.

In addition to supporting the methods and operations of mappings, *UserDict* instances provide the following attribute:

data

A real dictionary used to store the contents of the *UserDict* class.

8.3.8 `UserList` objects

This class acts as a wrapper around list objects. It is a useful base class for your own list-like classes which can inherit from them and override existing methods or add new ones. In this way, one can add new behaviors to lists.

The need for this class has been partially supplanted by the ability to subclass directly from *list*; however, this class can be easier to work with because the underlying list is accessible as an attribute.

class collections.**UserList**([*list*])

Class that simulates a list. The instance's contents are kept in a regular list, which is accessible via the *data* attribute of *UserList* instances. The instance's contents are initially set to a copy of *list*, defaulting to the empty list `[]`. *list* can be any iterable, for example a real Python list or a *UserList* object.

In addition to supporting the methods and operations of mutable sequences, *UserList* instances provide the following attribute:

data

A real *list* object used to store the contents of the *UserList* class.

Subclassing requirements: Subclasses of *UserList* are expected to offer a constructor which can be called with either no arguments or one argument. List operations which return a new sequence attempt to create an instance of the actual implementation class. To do so, it assumes that the constructor can be called with a single parameter, which is a sequence object used as a data source.

If a derived class does not wish to comply with this requirement, all of the special methods supported by this class will need to be overridden; please consult the sources for information about the methods which need to be provided in that case.

8.3.9 `UserString` objects

The class, *UserString* acts as a wrapper around string objects. The need for this class has been partially supplanted by the ability to subclass directly from *str*; however, this class can be easier to work with because the underlying string is accessible as an attribute.

class collections.**UserString**(*seq*)

Class that simulates a string object. The instance's content is kept in a regular string object, which is accessible via the *data* attribute of *UserString* instances. The instance's contents are initially set to a copy of *seq*. The *seq* argument can be any object which can be converted into a string using the built-in *str()* function.

In addition to supporting the methods and operations of strings, *UserString* instances provide the following attribute:

data

A real *str* object used to store the contents of the *UserString* class.

Alterado na versão 3.5: New methods `__getnewargs__`, `__rmod__`, `casefold`, `format_map`, `isprintable`, and `maketrans`.

8.4 `collections.abc` — Classes Base Abstratas para Contêineres

Novo na versão 3.3: Formerly, this module was part of the *collections* module. Anteriormente, esse módulo fazia parte do módelu *collections*.

Código Fonte: Lib/_collections_abc.py

Esse módulo fornece *classes base abstratas* que podem ser usadas para testar se uma classe fornece uma interface específica; por exemplo, se é hashable ou se é um mapeamento.

8.4.1 Coleções Abstratas Classes Base

O módulo de coleções oferece o seguinte *ABCs*:

ABC	Herda de	Métodos Abstratos	Métodos Mixin
Container		`__contains__`	
Hashable		`__hash__`	
Iterable		`__iter__`	
Iterator	*Iterable*	`__next__`	`__iter__`
Reversible	*Iterable*	`__reversed__`	
Generator	*Iterator*	`send`, `throw`	`close`, `__iter__`, `__next__`
Sized		`__len__`	
Callable		`__call__`	
Collection	*Sized*, *Iterable*, *Container*	`__contains__`, `__iter__`, `__len__`	
Sequence	*Reversible*, *Collection*	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index`, and `count`
MutableSequence	*Sequence*	`__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `insert`	Herda os métodos da *Sequence* e `append`, `reverse`, `extend`, `pop`, `remove`, e `__iadd__`
ByteString	*Sequence*	`__getitem__`, `__len__`	Herdado *Sequence* métodos
Set	*Collection*	`__contains__`, `__iter__`, `__len__`	`__le__`, `__lt__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__and__`, `__or__`, `__sub__`, `__xor__`, e `isdisjoint`
MutableSet	*Set*	`__contains__`, `__iter__`, `__len__`, `add`, `discard`	Herdado *Set* métodos e `clear`, `pop`, `remove`, `__ior__`, `__iand__`, `__ixor__`, e `__isub__`
Mapping	*Collection*	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `keys`, `items`, `values`, `get`, `__eq__`, e `__ne__`
MutableMapping	*Mapping*	`__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__`	Herdado *Mapping* métodos e `pop`, `popitem`, `clear`, `update`, e `setdefault`
MappingView	*Sized*		`__len__`
ItemsView	*MappingView*, *Set*		`__contains__`, `__iter__`
KeysView	*MappingView*, *Set*		`__contains__`, `__iter__`
ValuesView	*MappingView*, *Collection*		`__contains__`, `__iter__`
Awaitable		`__await__`	
Coroutine	*Awaitable*	`send`, `throw`	`close`
AsyncIterable		`__aiter__`	
AsyncIterator	*AsyncIterable*	`__anext__`	`__aiter__`
AsyncGenerator	*AsyncIterator*	`asend`, `athrow`	`aclose`, `__aiter__`, `__anext__`

class collections.abc.**Container**
class collections.abc.**Hashable**
class collections.abc.**Sized**
class collections.abc.**Callable**

ABCs para classes que fornecem respectivamente os métodos `__contains__()`, `__hash__()`, `__len__()`, e `__call__()`.

class collections.abc.**Iterable**

ABC para classes que fornecem o método `__iter__()`.

A verificação `isinstance(obj, Iterable)` detecta classes que são registradas como *Iterable* ou que possuem um método `__iter__()`, mas que não detecta classes que iteram com o método `__getitem__()`. A única maneira confiável de determinar se um objeto é *iterable* é chamar `iter(obj)`.

class collections.abc.**Collection**
ABC para classes de contêineres iteráveis de tamanho.

Novo na versão 3.6.

class collections.abc.**Iterator**
ABC para classes que fornecem os métodos *__iter__()* e métodos *__next__()*. Veja também a definição de *iterator*.

class collections.abc.**Reversible**
ABC para classes iteráveis que também fornecem o método `__reversed__()`.

Novo na versão 3.6.

class collections.abc.**Generator**
ABC para classes geradores que implementam o protocolo definido em **PEP 342** que estende os iteradores com os métodos `send()`, `throw()` e `close()`. Veja também a definição de *generator*.

Novo na versão 3.5.

class collections.abc.**Sequence**
class collections.abc.**MutableSequence**
class collections.abc.**ByteString**
ABCs para sequências somente de leitura e mutável :term:‘ <sequence>‘.

Nota de implementação: Alguns dos métodos mixin, como `__iter__()`, `__reversed__()` e `index()`, fazem chamadas repetidas para o método subjacente `__getitem__()`. Consequentemente, se `__getitem__()` for implementado com velocidade de acesso constante, os métodos mixin terão desempenho linear; no entanto se o método subjacente for linear (como seria com uma lista encadeada), os mixins terão desempenho quadrático e provavelmente precisarão ser substituídos.

Alterado na versão 3.5: O método index() adicionou suporte para os argumentos *stop* e *start*.

class collections.abc.**Set**
class collections.abc.**MutableSet**
ABCs para sets somente leitura e mutável.

class collections.abc.**Mapping**
class collections.abc.**MutableMapping**
ABCs para somente leitura e mutável *mappings*.

class collections.abc.**MappingView**
class collections.abc.**ItemsView**
class collections.abc.**KeysView**
class collections.abc.**ValuesView**
ABCs para mapeamento, itens, chaves e valores *views*.

class collections.abc.**Awaitable**
ABC para objetos *awaitable*, que podem ser usados em expressões de `await`. Implementações personalizadas devem fornecer o método `__await__()` method.

Coroutine objetos e instâncias do *Coroutine* ABC são todas instâncias dessa ABC.

Nota: No CPython, as corotinas baseados em gerador (geradoras decorados com *types.coroutine()* ou *asyncio.coroutine()*) são *awaitables*, embora não possuam o método `__await__()`. Usar

`isinstance(gencoro, Awaitable)` para eles retornará `False`. Use *`inspect.isawaitable()`* para detectá-los.

Novo na versão 3.5.

class collections.abc.**Coroutine**
ABC para classes compatíveis com coroutine. Eles implementam os seguintes métodos, definidos em coroutine-objects: `send()`, `throw()`, e `close()`. Implementações personalizadas também devem implementar `__await__()`. Todas as instâncias *`Coroutine`* também são instâncias de *`Awaitable`*. Veja também a definição de *coroutine*.

Nota: Em CPython, as corotinas baseadas em gerador (geradores decorados com *`types.coroutine()`* ou *`asyncio.coroutine()`*) são *awaitables*, embora não possuam o método `__await__()`. Usar `isinstance(gencoro, Coroutine)` para eles retornará `False`. Use *`inspect.isawaitable()`* para detectá-los.

Novo na versão 3.5.

class collections.abc.**AsyncIterable**
ABC para classes que fornecem o método `__aiter__`. Veja também a definição de *asynchronous iterable*.

Novo na versão 3.5.

class collections.abc.**AsyncIterator**
ABC para classes que fornecem os métodos `__aiter__` e `__anext__`. Veja também a definição de *asynchronous iterator*.

Novo na versão 3.5.

class collections.abc.**AsyncGenerator**
ABC para classes de gerador assíncrono que implementam o protocolo definido em **PEP 525** e **PEP 492**.

Novo na versão 3.6.

Esses ABCs nos permitem perguntar a classes ou instâncias se elas fornecem funcionalidades específicas, por exemplo:

```
size = None
if isinstance(myvar, collections.abc.Sized):
    size = len(myvar)
```

Vários ABCS também são úteis como mixins que facilitam o desenvolvimento de classes que suportam APIs de contêiner. Por exemplo, para escrever uma classe que suporte toda a API *`Set`* , é necessário fornecer apenas os três métodos abstratos subjacentes: `__contains__()`, `__iter__()`, e `__len__()`. O ABC fornece os métodos restantes, como `__and__()` e `isdisjoint()`:

```
class ListBasedSet(collections.abc.Set):
    ''' Alternate set implementation favoring space over speed
        and not requiring the set elements to be hashable. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)

    def __iter__(self):
        return iter(self.elements)
```

(continua na próxima página)

(continuação da página anterior)

```
    def __contains__(self, value):
        return value in self.elements

    def __len__(self):
        return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2            # The __and__() method is supported automatically
```

Notas sobre o uso de *Set* e *MutableSet* como um mixin:

(1) Como algumas operações de conjunto criam novos conjuntos, os métodos de mixin padrão precisam de uma maneira de criar novas instâncias a partir de uma iterável. Supõe-se que a classe construtor tenha uma assinatura no formato `ClassName(iterable)`. Essa suposição é fatorada em um método de classe interno chamado: `_from_iterable()` que chama `cls(iterable)` para produzir um novo conjunto. Se o mixin *Set* estiver sendo usado em uma classe com uma assinatura de construtor diferente, você precisará substituir `_from_iterable()` por um método de classe que possa construir novas instâncias a partir de um argumento iterável.

(2) Para substituir as comparações (presumivelmente para velocidade, já que a semântica é fixa), redefina `__le__()` e `__ge__()`, então as outras operações seguirão o exemplo automaticamente.

(3) O mixin *Set* fornece um método `_hash()` para calcular um valor de hash para o conjunto; no entanto, `__hash__()` não pe definido porque nem todos os conjuntos são encadeados ou imutáveis. Para adicionar capacidade de encadeamento em conjuntos usando mixin, herde de ambos *Set()* e *Hashable()*, e então defina `__hash__ = Set._hash`.

Ver também:

- OrderedSet receita para um exemplo baseado em *MutableSet*.
- Para mais informações sobre ABCs, consulte o módulo *abc* e **PEP 3119**.

8.5 `heapq` — Heap queue algorithm

Código Fonte: Lib/heapq.py

This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

Heaps are binary trees for which every parent node has a value less than or equal to any of its children. This implementation uses arrays for which `heap[k] <= heap[2*k+1]` and `heap[k] <= heap[2*k+2]` for all *k*, counting elements from zero. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that its smallest element is always the root, `heap[0]`.

The API below differs from textbook heap algorithms in two aspects: (a) We use zero-based indexing. This makes the relationship between the index for a node and the indexes for its children slightly less obvious, but is more suitable since Python uses zero-based indexing. (b) Our pop method returns the smallest item, not the largest (called a "min heap" in textbooks; a "max heap" is more common in texts because of its suitability for in-place sorting).

These two make it possible to view the heap as a regular Python list without surprises: `heap[0]` is the smallest item, and `heap.sort()` maintains the heap invariant!

To create a heap, use a list initialized to `[]`, or you can transform a populated list into a heap via function *heapify()*.

As seguintes funções são fornecidas:

heapq.**heappush**(*heap*, *item*)
Push the value *item* onto the *heap*, maintaining the heap invariant.

heapq.**heappop**(*heap*)
Pop and return the smallest item from the *heap*, maintaining the heap invariant. If the heap is empty, *IndexError* is raised. To access the smallest item without popping it, use `heap[0]`.

heapq.**heappushpop**(*heap*, *item*)
Push *item* on the heap, then pop and return the smallest item from the *heap*. The combined action runs more efficiently than *heappush()* followed by a separate call to *heappop()*.

heapq.**heapify**(*x*)
Transform list *x* into a heap, in-place, in linear time.

heapq.**heapreplace**(*heap*, *item*)
Pop and return the smallest item from the *heap*, and also push the new *item*. The heap size doesn't change. If the heap is empty, *IndexError* is raised.

This one step operation is more efficient than a *heappop()* followed by *heappush()* and can be more appropriate when using a fixed-size heap. The pop/push combination always returns an element from the heap and replaces it with *item*.

The value returned may be larger than the *item* added. If that isn't desired, consider using *heappushpop()* instead. Its push/pop combination returns the smaller of the two values, leaving the larger value on the heap.

The module also offers three general purpose functions based on heaps.

heapq.**merge**(**iterables*, *key=None*, *reverse=False*)
Merge multiple sorted inputs into a single sorted output (for example, merge timestamped entries from multiple log files). Returns an *iterator* over the sorted values.

Similar to `sorted(itertools.chain(*iterables))` but returns an iterable, does not pull the data into memory all at once, and assumes that each of the input streams is already sorted (smallest to largest).

Possui dois argumentos opcionais que devem ser especificados como argumentos nomeados.

key specifies a *key function* of one argument that is used to extract a comparison key from each input element. The default value is `None` (compare the elements directly).

reverse is a boolean value. If set to `True`, then the input elements are merged as if each comparison were reversed. To achieve behavior similar to `sorted(itertools.chain(*iterables), reverse=True)`, all iterables must be sorted from largest to smallest.

Alterado na versão 3.5: Added the optional *key* and *reverse* parameters.

heapq.**nlargest**(*n*, *iterable*, *key=None*)
Return a list with the *n* largest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). Equivalent to: `sorted(iterable, key=key, reverse=True)[:n]`.

heapq.**nsmallest**(*n*, *iterable*, *key=None*)
Return a list with the *n* smallest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). Equivalent to: `sorted(iterable, key=key)[:n]`.

The latter two functions perform best for smaller values of *n*. For larger values, it is more efficient to use the *sorted()* function. Also, when `n==1`, it is more efficient to use the built-in *min()* and *max()* functions. If repeated usage of these functions is required, consider turning the iterable into an actual heap.

8.5.1 Exemplos básicos

A heapsort can be implemented by pushing all values onto a heap and then popping off the smallest values one at a time:

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This is similar to `sorted(iterable)`, but unlike *`sorted()`*, this implementation is not stable.

Heap elements can be tuples. This is useful for assigning comparison values (such as task priorities) alongside the main record being tracked:

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

8.5.2 Priority Queue Implementation Notes

A priority queue is common use for a heap, and it presents several implementation challenges:

- Sort stability: how do you get two tasks with equal priorities to be returned in the order they were originally added?
- Tuple comparison breaks for (priority, task) pairs if the priorities are equal and the tasks do not have a default comparison order.
- If the priority of a task changes, how do you move it to a new position in the heap?
- Or if a pending task needs to be deleted, how do you find it and remove it from the queue?

A solution to the first two challenges is to store entries as 3-element list including the priority, an entry count, and the task. The entry count serves as a tie-breaker so that two tasks with the same priority are returned in the order they were added. And since no two entry counts are the same, the tuple comparison will never attempt to directly compare two tasks.

Another solution to the problem of non-comparable tasks is to create a wrapper class that ignores the task item and only compares the priority field:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

The remaining challenges revolve around finding a pending task and making changes to its priority or removing it entirely. Finding a task can be done with a dictionary pointing to an entry in the queue.

Removing the entry or changing its priority is more difficult because it would break the heap structure invariants. So, a possible solution is to mark the entry as removed and add a new entry with the revised priority:

```
pq = []                         # list of entries arranged in a heap
entry_finder = {}               # mapping of tasks to entries
REMOVED = '<removed-task>'      # placeholder for a removed task
counter = itertools.count()     # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    'Mark an existing task as REMOVED.  Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    'Remove and return the lowest priority task. Raise KeyError if empty.'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')
```

8.5.3 Teoria

Heaps are arrays for which `a[k] <= a[2*k+1]` and `a[k] <= a[2*k+2]` for all *k*, counting elements from 0. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that `a[0]` is always its smallest element.

The strange invariant above is meant to be an efficient memory representation for a tournament. The numbers below are *k*, not `a[k]`:

```
                                0

             1                                  2

      3              4                5               6

  7       8       9       10      11      12      13      14

15 16   17 18   19 20   21 22   23 24   25 26   27 28   29 30
```

In the tree above, each cell *k* is topping `2*k+1` and `2*k+2`. In a usual binary tournament we see in sports, each cell is the winner over the two cells it tops, and we can trace the winner down the tree to see all opponents s/he had. However, in many computer applications of such tournaments, we do not need to trace the history of a winner. To be more memory efficient, when a winner is promoted, we try to replace it by something else at a lower level, and the rule becomes that a cell and the two cells it tops contain three different items, but the top cell "wins" over the two topped cells.

If this heap invariant is protected at all time, index 0 is clearly the overall winner. The simplest algorithmic way to remove it and find the "next" winner is to move some loser (let's say cell 30 in the diagram above) into the 0 position, and then

percolate this new 0 down the tree, exchanging values, until the invariant is re-established. This is clearly logarithmic on the total number of items in the tree. By iterating over all items, you get an O(n log n) sort.

A nice feature of this sort is that you can efficiently insert new items while the sort is going on, provided that the inserted items are not "better" than the last 0'th element you extracted. This is especially useful in simulation contexts, where the tree holds all incoming events, and the "win" condition means the smallest scheduled time. When an event schedules other events for execution, they are scheduled into the future, so they can easily go into the heap. So, a heap is a good structure for implementing schedulers (this is what I used for my MIDI sequencer :-).

Various structures for implementing schedulers have been extensively studied, and heaps are good for this, as they are reasonably speedy, the speed is almost constant, and the worst case is not much different than the average case. However, there are other representations which are more efficient overall, yet the worst cases might be terrible.

Heaps are also very useful in big disk sorts. You most probably all know that a big sort implies producing "runs" (which are pre-sorted sequences, whose size is usually related to the amount of CPU memory), followed by a merging passes for these runs, which merging is often very cleverly organised[1]. It is very important that the initial sort produces the longest runs possible. Tournaments are a good way to achieve that. If, using all the memory available to hold a tournament, you replace and percolate items that happen to fit the current run, you'll produce runs which are twice the size of the memory for random input, and much better for input fuzzily ordered.

Moreover, if you output the 0'th item on disk and get an input which may not fit in the current tournament (because the value "wins" over the last output value), it cannot fit in the heap, so the size of the heap decreases. The freed memory could be cleverly reused immediately for progressively building a second heap, which grows at exactly the same rate the first heap is melting. When the first heap completely vanishes, you switch heaps and start a new run. Clever and quite effective!

In a word, heaps are useful memory structures to know. I use them in a few applications, and I think it is good to keep a 'heap' module around. :-)

8.6 `bisect` — Algoritmo de bisseção de vetor

Código Fonte: Lib/bisect.py

Este módulo fornece suporte para manter uma lista em ordem de classificação sem ter que classificar a lista após cada inserção. Para longas listas de itens com operações de comparação custosas, isso pode ser uma melhoria em relação à abordagem mais comum. O módulo é denominado `bisect` porque usa um algoritmo de bisseção básico para fazer seu trabalho. O código-fonte pode ser mais útil como um exemplo funcional de algoritmo (as condições fronteiriças já estão certas!).

As seguintes funções são fornecidas:

`bisect.`**`bisect_left`**(*a*, *x*, *lo=0*, *hi=len(a)*)

Localiza o ponto de inserção de *x* em *a* para manter a ordem de classificação. Os parâmetros *lo* e *hi* podem ser usados para especificar um subconjunto da lista que deve ser considerado; por padrão, toda a lista é usada. Se *x* já estiver presente em *a*, o ponto de inserção estará antes (à esquerda) de qualquer entrada existente. O valor de retorno é adequado para uso como o primeiro parâmetro para `list.insert()` supondo que *a* já esteja ordenado.

O ponto de inserção retornado *i* particiona o vetor *a* em duas metades de modo que `all(val < x for val in a[lo:i])` para o lado esquerdo e `all(val >= x for val in a[i:hi])` para o lado direito.

`bisect.`**`bisect_right`**(*a*, *x*, *lo=0*, *hi=len(a)*)

[1] The disk balancing algorithms which are current, nowadays, are more annoying than clever, and this is a consequence of the seeking capabilities of the disks. On devices which cannot seek, like big tape drives, the story was quite different, and one had to be very clever to ensure (far in advance) that each tape movement will be the most effective possible (that is, will best participate at "progressing" the merge). Some tapes were even able to read backwards, and this was also used to avoid the rewinding time. Believe me, real good tape sorts were quite spectacular to watch! From all times, sorting has always been a Great Art! :-)

bisect.**bisect**(*a*, *x*, *lo=0*, *hi=len(a)*)
Semelhante a *bisect_left()*, mas retorna um ponto de inserção que vem depois (à direita de) qualquer entrada existente de *x* em *a*.

O ponto de inserção retornado *i* particiona o vetor *a* em duas metades de modo que `all(val <= x for val in a[lo:i])` para o lado esquerdo e `all(val > x for val in a[i:hi])` para o lado direito.

bisect.**insort_left**(*a*, *x*, *lo=0*, *hi=len(a)*)
Insere *x* em *a* na ordem de classificação. Isso é equivalente a `a.insert(bisect.bisect_left(a, x, lo, hi), x)` supondo que *a* já esteja ordenado. Lembre-se de que a pesquisa O(log n) é dominada pela lenta etapa de inserção O(n).

bisect.**insort_right**(*a*, *x*, *lo=0*, *hi=len(a)*)
bisect.**insort**(*a*, *x*, *lo=0*, *hi=len(a)*)
Semelhante a *insort_left()*, mas inserindo *x* em *a* após qualquer entrada existente de *x*.

Ver também:

Receita de SortedCollection que usa bisect para construir uma classe de coleção completa com métodos de pesquisa diretos e suporte para uma função chave. As chaves são pré-calculadas para economizar em chamadas desnecessárias para a função chave durante as pesquisas.

8.6.1 Pesquisando em listas ordenadas

As funções *bisect()* acima são úteis para encontrar pontos de inserção, mas podem ser complicadas ou difíceis de usar para tarefas comuns de pesquisa. As cinco funções a seguir mostram como transformá-las nas pesquisas padrão para listas ordenadas:

```
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
```

(continua na próxima página)

(continuação da página anterior)

```
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

8.6.2 Outros Exemplos

A função *bisect()* pode ser útil para pesquisas em tabelas numéricas. Este exemplo usa *bisect()* para pesquisar uma nota em letra para uma pontuação de exame (digamos) com base em um conjunto de pontos de interrupção numéricos ordenados: 90 e acima é um "A", 80 a 89 é um "B" e por aí vai:

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
...     i = bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

Ao contrário da função *sorted()*, não faz sentido que as funções *bisect()* tenham argumentos *key* ou *reversed* porque isso levaria a um design ineficiente (chamadas sucessivas para funções bisect não "lembrariam" de todas as pesquisas de chave anteriores).

Em vez disso, é melhor pesquisar uma lista de chaves pré-computadas para encontrar o índice do registro em questão:

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])
>>> keys = [r[1] for r in data]         # precomputed list of keys
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

8.7 `array`— Arrays eficientes de valores numéricos

Esse módulo define um tipo de objeto que pode representar compactamente um array de valores básicos: caracteres, inteiros, números de ponto flutuante. Arrays são tipos de sequência e funcionam bem parecidamente com listas, porém o tipo dos objetos armazenados é restringido. O tipo é especificado na criação do objeto usando um type code, que é um único caractere. Os seguintes type codes são definidos:

Type code	Tipo em C	Tipo em Python	Tamanho mínimo em bytes	Notas
'b'	signed char	int	1	
'B'	unsigned char	int	1	
'u'	Py_UNICODE	Caractere unicode	2	(1)
'h'	signed short	int	2	
'H'	unsigned short	int	2	
'i'	signed int	int	2	
'I'	unsigned int	int	2	
'l'	signed long	int	4	
'L'	unsigned long	int	4	
'q'	signed long long	int	8	
'Q'	unsigned long long	int	8	
'f'	float	float	4	
'd'	double	float	8	

Notas:

(1) O 'u' corresponde ao obsoleto caractere unicode (Py_UNICODE que é wchar_t). Dependendo da plataforma, pode ser de 16 bits ou 32 bits.

O 'u' será removido junto com o resto da Py_UNICODE API.

Deprecated since version 3.3, will be removed in version 4.0.

A representação dos valores é definida pela arquitetura da máquina, mais especificamente da implementação do C. O tamanho real pode ser acessado pelo atributo itemsize

O módulo define o seguinte tipo:

class array.**array**(*typecode*[, *initializer*])

Um novo array cujos itens são restritos pelo *type code* e inicializados pelo valor opcional *initializer*, que deve ser uma lista, um *bytes-like object*, ou outro elementos iterável do tipo apropriado.

Se passado uma lista ou string, o inicializador é passado para os métodos *fromlist()*, *frombytes()*, ou *fromunicode()* (ver abaixo) do novo array para adicionar itens iniciais ao array. Caso contrário, o inicializador iterável é passado para o método *extend()*.

array.**typecodes**

String com todos os types codes disponíveis.

Objetos array tem suporte para as operações de sequência comuns: indexação, fatiamento, concatenação, e multiplicação. Quando usando a atribuição de fatias, o valor associado deve ser um objeto array com o mesmo código de tipo; caso contrário , *TypeError* é levantada. Objetos array também implementam a interface buffer, e também podem ser usados em qualquer lugar onde *bytes-like objects* é permitido.

Os seguintes itens e métodos também são suportados:

array.**typecode**

O caractere typecode usado para criar o array.

array.**itemsize**

O tamanho em bytes de um item do array em representação interna.

array.**append**(*x*)

Adiciona um novo item com valor *x* ao final do array.

array.**buffer_info**()

Retorna uma tupla (address, length) com o endereço corrente da memória e o tamanho em elementos do buffer usado para armazenar conteúdos do array. O tamanho do buffer da memória em bytes pode ser computado como array.buffer_info()[1] * array.itemsize. Isso é ocasionalmente útil quando se está

trabalhando com interfaces I/O de baixo nível (inerentemente inseguras) que precisam de endereços de memória, como algumas operações `ioctl()`. Os números retornados são válidos enquanto o array existir e nenhuma operação de alteração de tamanho for aplicada a ele.

Nota: Quando se está usando arrays de código escrito em C ou C++ (o único jeito efetivo de usar essa informação), faz mais sentido usar a interface do buffer suportada pelos arrays. Esse método é mantido para retrocompatibilidade e deve ser evitado em código novo. A interface de buffers está documentada em bufferobjects.

`array.`**`byteswap`**`()`
"Byteswap" todos os itens do vetor. Isso é somente suportado para valores de 1, 2, 4 ou 8 bytes de tamanho; para outros tipos de valores é levantada *RuntimeError*. Isso é útil quando estamos lendo dados de um arquivo para serem escritos em um arquivo de outra máquina de ordem de bytes diferente.

`array.`**`count`**`(`*x*`)`
Retorna a quantidade de ocorrências de *x* no array.

`array.`**`extend`**`(`*iterable*`)`
Acrescenta os itens em *iterable* ao final do array. Se *iterable* for outro array, ele deve ter *exatamente* o mesmo type code; senão, ocorrerá um *TypeError*. Se *iterable* não for um array, ele deve ser iterável e seus elementos devem ser do tipo correto para ser acrescentado ao array,

`array.`**`frombytes`**`(`*s*`)`
Adiciona itens da string, interpretando a string como um array (como se tivesse sido lido de um arquivo usando o método *fromfile()*).

Novo na versão 3.2: *fromstring()* for renomeado para *frombytes()* para maior clareza.

`array.`**`fromfile`**`(`*f*, *n*`)`
Lê *n* itens (como valores de máquinas) do *objeto arquivo* *f* e adiciona-os ao fim do vetor. Se estão disponíveis menos de *n* itens, *EOFError* é levantada, mas os itens disponíveis ainda são inseridos ao final do vetor. *f* deve realmente ser um objeto arquivo; qualquer outra coisa com um método `read()` não vai funcionar.

`array.`**`fromlist`**`(`*list*`)`
Adiciona itens de *list*. Isso é equivalente a `for x in list: a.append(x)` exceto que se ocorrer um errro de tipo, o vetor não é alterado.

`array.`**`fromstring`**`()`
Alias deprecado para *frombytes()*.

Deprecated since version 3.2, will be removed in version 3.9.

`array.`**`fromunicode`**`(`*s*`)`
Estende este vetor com os dados da strind unicode fornecida. O vetor deve ser do tipo `'u'`; aso contrário uma *ValueError* será levantada. Use `array.frombytes(unicodestring.encode(enc))` para adicionar dados Unicode para um vetor de outros tipos de dados.

`array.`**`index`**`(`*x*`)`
Retorna o menor *i* em que *i* é o índice da primeira ocorrência de *x* no vetor.

`array.`**`insert`**`(`*i*, *x*`)`
Insere um novo item com o *x* no vetor antes da posição *i*. Valores negativos são tratados como sendo em relação ao fim do vetor.

`array.`**`pop`**`(`[*i*]`)`
Remove o item com o índice *i* do vetor e retorna este item. O valor padrão do argumento é `-1`, assim por padrão o último item é removido e retornado.

`array.`**`remove`**`(`*x*`)`
Remove a primeira ocorrência de *x* da array.

`array.reverse()`
Inverte a ordem dos itens na array.

`array.tobytes()`
Devolve os itens do vetor como um vetor de valores de máquina com a representação em bytes (a mesma sequência de bytes que seria escrita pelo método *tofile()*.)

Novo na versão 3.2: *tostring()* foi nomeada para *tobytes()* para maior clareza.

`array.tofile(f)`
Escreve todos os itens (como valores de máquinas) para o *objeto arquivo* *f*.

`array.tolist()`
Converte a array para uma lista comum com os mesmos itens.

`array.tostring()`
Deprecated alias for *tobytes()*.

Deprecated since version 3.2, will be removed in version 3.9.

`array.tounicode()`
Devolve os itens do vetor como uma string unicode. O vetor deve ser do tipo `'u'`; caso contrário *ValueError* é levantada. Use `array.tobytes().decode(enc)` para obter uma string unicode de um vetor de outros tipos.

Quando um vetor é exibido ou convertido para uma string, é representado como `array(typecode, initializer)`. O *initializer* é omitido se o vetor estiver vazio, caso contrário será uma string se *typecode* for `'u'`, se não será uma lista de números. É garantido que será possível uma conversão da string de volta para um vetor com o mesmo tipo e valor usando *eval()*, contanto que a classe *array* tenha sido importada usando `from array import array`. Exemplos:

```
array('l')
array('u', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

Ver também:

Modulo *struct* Empacotamento e desempacotamento de dados binários heterogêneos.

Módulo *xdrlib* Empacotamento e desempacotamento de dados External Data Representation (XDR) usados em alguns sistemas para chamada remota de procedimentos.

The Numerical Python Documentation A extensão Numeric Python (NumPy) define outro tipo array; veja http://www.numpy.org/ para mais informações sobre Numerical Python.

8.8 `weakref` — Weak references

Código Fonte: Lib/weakref.py

The *weakref* module allows the Python programmer to create *weak references* to objects.

In the following, the term *referent* means the object which is referred to by a weak reference.

A weak reference to an object is not enough to keep the object alive: when the only remaining references to a referent are weak references, *garbage collection* is free to destroy the referent and reuse its memory for something else. However, until the object is actually destroyed the weak reference may return the object even if there are no strong references to it.

A primary use for weak references is to implement caches or mappings holding large objects, where it's desired that a large object not be kept alive solely because it appears in a cache or mapping.

For example, if you have a number of large binary image objects, you may wish to associate a name with each. If you used a Python dictionary to map names to images, or images to names, the image objects would remain alive just because they appeared as values or keys in the dictionaries. The *WeakKeyDictionary* and *WeakValueDictionary* classes supplied by the *weakref* module are an alternative, using weak references to construct mappings that don't keep objects alive solely because they appear in the mapping objects. If, for example, an image object is a value in a *WeakValueDictionary*, then when the last remaining references to that image object are the weak references held by weak mappings, garbage collection can reclaim the object, and its corresponding entries in weak mappings are simply deleted.

WeakKeyDictionary and *WeakValueDictionary* use weak references in their implementation, setting up callback functions on the weak references that notify the weak dictionaries when a key or value has been reclaimed by garbage collection. *WeakSet* implements the *set* interface, but keeps weak references to its elements, just like a *WeakKeyDictionary* does.

finalize provides a straight forward way to register a cleanup function to be called when an object is garbage collected. This is simpler to use than setting up a callback function on a raw weak reference, since the module automatically ensures that the finalizer remains alive until the object is collected.

Most programs should find that using one of these weak container types or *finalize* is all they need – it's not usually necessary to create your own weak references directly. The low-level machinery is exposed by the *weakref* module for the benefit of advanced uses.

Not all objects can be weakly referenced; those objects which can include class instances, functions written in Python (but not in C), instance methods, sets, frozensets, some *file objects*, *generators*, type objects, sockets, arrays, deques, regular expression pattern objects, and code objects.

Alterado na versão 3.2: Added support for thread.lock, threading.Lock, and code objects.

Several built-in types such as *list* and *dict* do not directly support weak references but can add support through subclassing:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # this object is weak referenceable
```

CPython implementation detail: Other built-in types such as *tuple* and *int* do not support weak references even when subclassed.

Extension types can easily be made to support weak references; see weakref-support.

class weakref.**ref**(*object*[, *callback*])

Return a weak reference to *object*. The original object can be retrieved by calling the reference object if the referent is still alive; if the referent is no longer alive, calling the reference object will cause *None* to be returned. If *callback* is provided and not *None*, and the returned weakref object is still alive, the callback will be called when the object is about to be finalized; the weak reference object will be passed as the only parameter to the callback; the referent will no longer be available.

It is allowable for many weak references to be constructed for the same object. Callbacks registered for each weak reference will be called from the most recently registered callback to the oldest registered callback.

Exceptions raised by the callback will be noted on the standard error output, but cannot be propagated; they are handled in exactly the same way as exceptions raised from an object's `__del__()` method.

Weak references are *hashable* if the *object* is hashable. They will maintain their hash value even after the *object* was deleted. If *hash()* is called the first time only after the *object* was deleted, the call will raise *TypeError*.

Weak references support tests for equality, but not ordering. If the referents are still alive, two references have the same equality relationship as their referents (regardless of the *callback*). If either referent has been deleted, the references are equal only if the reference objects are the same object.

This is a subclassable type rather than a factory function.

__callback__
This read-only attribute returns the callback currently associated to the weakref. If there is no callback or if the referent of the weakref is no longer alive then this attribute will have value `None`.

Alterado na versão 3.4: Added the *__callback__* attribute.

`weakref.`**`proxy`**`(`*object*[, *callback*]`)`
Return a proxy to *object* which uses a weak reference. This supports use of the proxy in most contexts instead of requiring the explicit dereferencing used with weak reference objects. The returned object will have a type of either `ProxyType` or `CallableProxyType`, depending on whether *object* is callable. Proxy objects are not *hashable* regardless of the referent; this avoids a number of problems related to their fundamentally mutable nature, and prevent their use as dictionary keys. *callback* is the same as the parameter of the same name to the *ref()* function.

`weakref.`**`getweakrefcount`**`(`*object*`)`
Return the number of weak references and proxies which refer to *object*.

`weakref.`**`getweakrefs`**`(`*object*`)`
Return a list of all weak reference and proxy objects which refer to *object*.

class `weakref.`**`WeakKeyDictionary`**`(`[*dict*]`)`
Mapping class that references keys weakly. Entries in the dictionary will be discarded when there is no longer a strong reference to the key. This can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.

WeakKeyDictionary objects have an additional method that exposes the internal references directly. The references are not guaranteed to be "live" at the time they are used, so the result of calling the references needs to be checked before being used. This can be used to avoid creating references that will cause the garbage collector to keep the keys around longer than needed.

`WeakKeyDictionary.`**`keyrefs`**`()`
Return an iterable of the weak references to the keys.

class `weakref.`**`WeakValueDictionary`**`(`[*dict*]`)`
Mapping class that references values weakly. Entries in the dictionary will be discarded when no strong reference to the value exists any more.

WeakValueDictionary objects have an additional method that has the same issues as the `keyrefs()` method of *WeakKeyDictionary* objects.

`WeakValueDictionary.`**`valuerefs`**`()`
Return an iterable of the weak references to the values.

class `weakref.`**`WeakSet`**`(`[*elements*]`)`
Set class that keeps weak references to its elements. An element will be discarded when no strong reference to it exists any more.

class `weakref.`**`WeakMethod`**`(`*method*`)`
A custom *ref* subclass which simulates a weak reference to a bound method (i.e., a method defined on a class and looked up on an instance). Since a bound method is ephemeral, a standard weak reference cannot keep hold of it. *WeakMethod* has special code to recreate the bound method until either the object or the original function dies:

```
>>> class C:
...     def method(self):
...         print("method called!")
...
>>> c = C()
>>> r = weakref.ref(c.method)
>>> r()
>>> r = weakref.WeakMethod(c.method)
>>> r()
<bound method C.method of <__main__.C object at 0x7fc859830220>>
>>> r()()
method called!
>>> del c
>>> gc.collect()
0
>>> r()
>>>
```

Novo na versão 3.4.

class weakref.**finalize**(*obj*, *func*, **args*, ***kwargs*)

Return a callable finalizer object which will be called when *obj* is garbage collected. Unlike an ordinary weak reference, a finalizer will always survive until the reference object is collected, greatly simplifying lifecycle management.

A finalizer is considered *alive* until it is called (either explicitly or at garbage collection), and after that it is *dead*. Calling a live finalizer returns the result of evaluating `func(*arg, **kwargs)`, whereas calling a dead finalizer returns *None*.

Exceptions raised by finalizer callbacks during garbage collection will be shown on the standard error output, but cannot be propagated. They are handled in the same way as exceptions raised from an object's `__del__()` method or a weak reference's callback.

When the program exits, each remaining live finalizer is called unless its *atexit* attribute has been set to false. They are called in reverse order of creation.

A finalizer will never invoke its callback during the later part of the *interpreter shutdown* when module globals are liable to have been replaced by *None*.

__call__()

If *self* is alive then mark it as dead and return the result of calling `func(*args, **kwargs)`. If *self* is dead then return *None*.

detach()

If *self* is alive then mark it as dead and return the tuple `(obj, func, args, kwargs)`. If *self* is dead then return *None*.

peek()

If *self* is alive then return the tuple `(obj, func, args, kwargs)`. If *self* is dead then return *None*.

alive

Property which is true if the finalizer is alive, false otherwise.

atexit

A writable boolean property which by default is true. When the program exits, it calls all remaining live finalizers for which *atexit* is true. They are called in reverse order of creation.

Nota: It is important to ensure that *func*, *args* and *kwargs* do not own any references to *obj*, either directly or indirectly, since otherwise *obj* will never be garbage collected. In particular, *func* should not be a bound method

of *obj*.

Novo na versão 3.4.

`weakref.`**`ReferenceType`**
The type object for weak references objects.

`weakref.`**`ProxyType`**
The type object for proxies of objects which are not callable.

`weakref.`**`CallableProxyType`**
The type object for proxies of callable objects.

`weakref.`**`ProxyTypes`**
Sequence containing all the type objects for proxies. This can make it simpler to test if an object is a proxy without being dependent on naming both proxy types.

Ver também:

PEP 205 - Weak References The proposal and rationale for this feature, including links to earlier implementations and information about similar features in other languages.

8.8.1 Objetos de Referência Fraca

Weak reference objects have no methods and no attributes besides *ref.__callback__*. A weak reference object allows the referent to be obtained, if it still exists, by calling it:

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

If the referent no longer exists, calling the reference object returns *None*:

```
>>> del o, o2
>>> print(r())
None
```

Testing that a weak reference object is still live should be done using the expression `ref() is not None`. Normally, application code that needs to use a reference object should follow this pattern:

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print("Object has been deallocated; can't frobnicate.")
else:
    print("Object is still live!")
    o.do_something_useful()
```

Using a separate test for "liveness" creates race conditions in threaded applications; another thread can cause a weak reference to become invalidated before the weak reference is called; the idiom shown above is safe in threaded applications as well as single-threaded applications.

Specialized versions of *ref* objects can be created through subclassing. This is used in the implementation of the *WeakValueDictionary* to reduce the memory overhead for each entry in the mapping. This may be most useful to associate additional information with a reference, but could also be used to insert additional processing on calls to retrieve the referent.

This example shows how a subclass of *ref* can be used to store additional information about an object and affect the value that's returned when the referent is accessed:

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, **annotations):
        super(ExtendedRef, self).__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of
        times the reference has been called.
        """
        ob = super(ExtendedRef, self).__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

8.8.2 Exemplo

This simple example shows how an application can use object IDs to retrieve objects that it has seen before. The IDs of the objects can then be used in other data structures without forcing the objects to remain alive, but the objects can still be retrieved by ID if they do.

```
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```

8.8.3 Objetos finalizadores

The main benefit of using *finalize* is that it makes it simple to register a callback without needing to preserve the returned finalizer object. For instance

```
>>> import weakref
>>> class Object:
...     pass
...
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny!")
<finalize object at ...; for 'Object' at ...>
>>> del kenny
You killed Kenny!
```

The finalizer can be called directly as well. However the finalizer will invoke the callback at most once.

```
>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> assert f.alive
>>> assert f() == 6
CALLBACK
>>> assert not f.alive
>>> f()                     # callback not called because finalizer dead
>>> del obj                 # callback not called because finalizer dead
```

You can unregister a finalizer using its *detach()* method. This kills the finalizer and returns the arguments passed to the constructor when it was created.

```
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> f.detach()
(<...Object object ...>, <function callback ...>, (1, 2), {'z': 3})
>>> newobj, func, args, kwargs = _
>>> assert not f.alive
>>> assert newobj is obj
>>> assert func(*args, **kwargs) == 6
CALLBACK
```

Unless you set the *atexit* attribute to *False*, a finalizer will be called when the program exits if it is still alive. For instance

```
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")
<finalize object at ...; for 'Object' at ...>
>>> exit()
obj dead or exiting
```

8.8.4 Comparing finalizers with `__del__()` methods

Suppose we want to create a class whose instances represent temporary directories. The directories should be deleted with their contents when the first of the following events occurs:

- the object is garbage collected,
- the object's `remove()` method is called, or
- o programa finaliza.

We might try to implement the class using a `__del__()` method as follows:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()

    def remove(self):
        if self.name is not None:
            shutil.rmtree(self.name)
            self.name = None

    @property
    def removed(self):
        return self.name is None

    def __del__(self):
        self.remove()
```

Starting with Python 3.4, `__del__()` methods no longer prevent reference cycles from being garbage collected, and module globals are no longer forced to `None` during *interpreter shutdown*. So this code should work without any issues on CPython.

However, handling of `__del__()` methods is notoriously implementation specific, since it depends on internal details of the interpreter's garbage collector implementation.

A more robust alternative can be to define a finalizer which only references the specific functions and objects that it needs, rather than having access to the full state of the object:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()
        self._finalizer = weakref.finalize(self, shutil.rmtree, self.name)

    def remove(self):
        self._finalizer()

    @property
    def removed(self):
        return not self._finalizer.alive
```

Defined like this, our finalizer only receives a reference to the details it needs to clean up the directory appropriately. If the object never gets garbage collected the finalizer will still be called at exit.

The other advantage of weakref based finalizers is that they can be used to register finalizers for classes where the definition is controlled by a third party, such as running code when a module is unloaded:

```
import weakref, sys
def unloading_module():
```

(continua na próxima página)

(continuação da página anterior)

```
    # implicit reference to the module globals from the function body
weakref.finalize(sys.modules[__name__], unloading_module)
```

Nota: If you create a finalizer object in a daemonic thread just as the program exits then there is the possibility that the finalizer does not get called at exit. However, in a daemonic thread `atexit.register()`, `try: ... finally: ...` and `with: ...` do not guarantee that cleanup occurs either.

8.9 `types` — Criação de tipos dinâmicos e nomes para tipos embutidos

Código Fonte: Lib/types.py

Este módulo define funções utilitárias para auxiliar na criação dinâmica de novos tipos.

Também define nomes para alguns tipos de objetos usados pelo interpretador Python padrão, mas não expostos como componentes embutidos como `int` ou `str` são.

Por fim, fornece algumas classes e funções adicionais relacionadas ao tipo que não são fundamentais o suficiente para serem incorporadas.

8.9.1 Criação de Tipo Dinâmico

`types.`**`new_class`**(*name*, *bases=()*, *kwds=None*, *exec_body=None*)

Cria um objeto de classe dinamicamente usando a metaclasse apropriada.

Os três primeiros argumentos são os componentes que compõem um cabeçalho de definição de classe: o nome da classe, as classes base (em ordem), os argumentos nomeados (como `metaclass`).

O argumento *exec_body* é um retorno de chamada usado para preencher o espaço para nome da classe recém-criado. Ele deve aceitar o espaço para nome da classe como seu único argumento e atualizar o espaço para nome diretamente com o conteúdo da classe. Se nenhum retorno de chamada for fornecido, ele terá o mesmo efeito que passar em `lambda ns: ns`.

Novo na versão 3.3.

`types.`**`prepare_class`**(*name*, *bases=()*, *kwds=None*)

Calcula a metaclasse apropriada e cria o espaço de nomes da classe.

Os argumentos são os componentes que compõem um cabeçalho de definição de classe: o nome da classe, as classes base (em ordem) e os argumentos nomeados (como `metaclass`).

O valor de retorno é uma tupla de três: `metaclass, namespace, kwds`

metaclass é a metaclasse apropriada, *namespace* é o espaço de nomes da classe preparada e *kwds* é uma cópia atualizada do argumento passado no *kwds* com qualquer entrada `'metaclass'` removida. Se nenhum argumento *kwds* for passado, este será um ditado vazio.

Novo na versão 3.3.

Alterado na versão 3.6: O valor padrão para o elemento `namespace` da tupla retornada foi alterado. Agora, um mapeamento de preservação da ordem de inserção é usado quando a metaclasse não possui um método `__prepare__`.

Ver também:

metaclasses Detalhes completos do processo de criação de classe suportado por essas funções

PEP 3115 - Metaclasses no Python 3000 Introduzido o gancho de espaço de nomes `__prepare__`

`types.`**`resolve_bases`**(*bases*)
Resolve entradas MRO dinamicamente, conforme especificado por **PEP 560**.

Esta função procura por itens em *bases* que não sejam instâncias de *type* e retorna uma tupla onde cada objeto que possui um método `__mro_entries__` é substituído por um resultado descompactado da chamada desse método. Se um item *bases* é uma instância de *type*, ou não possui o método `__mro_entries__`, ele é incluído na tupla de retorno inalterada.

Novo na versão 3.7.

Ver também:

PEP 560 - Suporte básico para inserir módulo e tipos genéricos

8.9.2 Tipos padrão do intepretador

Este módulo fornece nomes para muitos dos tipos necessários para implementar um interpretador Python. Evita deliberadamente incluir alguns dos tipos que surgem apenas incidentalmente durante o processamento, como o tipo `listiterator`.

O uso típico desses nomes é para verificações *isinstance()* ou : func:*issubclass*.

Os nomes padrão são definidos para os seguintes tipos:

`types.`**`FunctionType`**
`types.`**`LambdaType`**
O tipo de funções definidas pelo usuário e funções criadas por expressões `lambda`.

`types.`**`GeneratorType`**
O tipo de objetos de iterador *gerador*, criados pelas funções geradoras.

`types.`**`CoroutineType`**
O tipo de objetos de *coroutine*, criado por funções de `async def`.

Novo na versão 3.5.

`types.`**`AsyncGeneratorType`**
O tipo de objetos de iterador *gerador assíncrono*, criados pelas funções do gerador assíncrono.

Novo na versão 3.6.

`types.`**`CodeType`**
O tipo de objetos de código retornados por *compile()*.

`types.`**`MethodType`**
O tipo de método de instâncias de classe definidas pelo usuário.

`types.`**`BuiltinFunctionType`**
`types.`**`BuiltinMethodType`**
O tipo de funções embutidas como *len()* ou *sys.exit()*, e métodos de classes embutidas. (Aqui, o termo "embutidas" significa "escrito em C".)

`types.`**`WrapperDescriptorType`**
O tipo de método de alguns tipos de dados embutidos e classes base, como `object.__init__()` ou `object.__lt__()`.

Novo na versão 3.7.

types.**MethodWrapperType**
O tipo de métodos *vinculados* de alguns tipos de dados embutidos e classes base. Por exemplo, é o tipo de `object().__str__`.

Novo na versão 3.7.

types.**MethodDescriptorType**
O tipo de método de alguns tipos de dados embutidos, como *str.join()*.

Novo na versão 3.7.

types.**ClassMethodDescriptorType**
O tipo de métodos de classe *não vinculados* de alguns tipos de dados embutidos, como `dict.__dict__['fromkeys']`.

Novo na versão 3.7.

class types.**ModuleType**(*name*, *doc=None*)
O tipo de *módulos*. O construtor recebe o nome do módulo a ser criado e opcionalmente sua *docstring*.

Nota: Use *importlib.util.module_from_spec()* para criar um novo módulo se você deseja definir os vários atributos controlados por importação.

__doc__
A *docstring* do módulo. O padrão é `None`.

__loader__
O *carregador* que carregou o módulo. O padrão é `None`.

Alterado na versão 3.4: O padrão é `None`. Anteriormente, o atributo era opcional.

__name__
O nome do módulo.

__package__
A qual *pacote* um módulo pertence. Se o módulo é de nível superior (ou seja, não faz parte de nenhum pacote específico), o atributo deve ser definido como `''`, senão deve ser definido como o nome do pacote (que pode ser `__name__` se o módulo for o próprio pacote). O padrão é `None`.

Alterado na versão 3.4: O padrão é `None`. Anteriormente, o atributo era opcional.

class types.**TracebackType**(*tb_next*, *tb_frame*, *tb_lasti*, *tb_lineno*)
O tipo de objetos traceback, como encontrados em `sys.exc_info()[2]`.

Veja a referência de linguagem para detalhes dos atributos e operações disponíveis, e orientação sobre como criar tracebacks dinamicamente.

types.**FrameType**
O tipo de objetos quadro, como encontrado em `tb.tb_frame` se `tb` é um objeto traceback.

Veja a referência de linguagem para detalhes dos atributos e operações disponíveis.

types.**GetSetDescriptorType**
O tipo de objetos definidos em módulos de extensão com `PyGetSetDef`, como `FrameType.f_locals` ou `array.array.typecode`. Este tipo é usado como descritor para atributos de objeto; tem o mesmo propósito que o tipo *property*, mas para classes definidas em módulos de extensão.

types.**MemberDescriptorType**
O tipo de objetos definidos em módulos de extensão com `PyMemberDef`, como `datetime.timedelta.days`. Este tipo é usado como descritor para membros de dados C simples que usam funções de conversão padrão; tem o mesmo propósito que o tipo *property*, mas para classes definidas em módulos de extensão.

CPython implementation detail: Em outras implementações de Python, este tipo pode ser idêntico a `GetSetDescriptorType`.

class types.**MappingProxyType**(*mapping*)
Proxy somente leitura de um mapeamento. Ele fornece uma visão dinâmica das entradas do mapeamento, o que significa que quando o mapeamento muda, a visão reflete essas mudanças.

Novo na versão 3.3.

key in proxy
Retorna `True` se o mapeamento subjacente tiver uma chave *key*, senão `False`.

proxy[key]
Retorna e o item do mapeamento subjacente com a chave *key*. Levanta um *KeyError* se *key* não estiver no mapeamento subjacente.

iter(proxy)
Retorna um iterador sobre as chaves do mapeamento subjacente. Este é um atalho para `iter(proxy.keys())`.

len(proxy)
Retorna o número de itens no mapeamento subjacente.

copy()
Retorna uma cópia rasa do mapeamento subjacente.

get(*key*[, *default*])
Retorna o valor para *key* se *key* estiver no mapeamento subjacente, caso contrário, *default*. Se *default* não for fornecido, o padrão é `None`, de forma que este método nunca levante uma *KeyError*.

items()
Retorna uma nova visão dos itens do mapeamento subjacente (pares `(chave, valor)`).

keys()
Retorna uma nova visão das chaves do mapeamento subjacente.

values()
Retorna uma nova visão dos valores do mapeamento subjacente.

8.9.3 Classes e funções de utilidades adicionais

class types.**SimpleNamespace**
Uma subclasse *object* simples que fornece acesso de atributo ao seu espaço de nomes, bem como um repr significativo.

Diferentemente de *object*, com `SimpleNamespace` você pode adicionar e remover atributos. Se um objeto `SimpleNamespace` é inicializado com argumentos nomeados, eles são adicionados diretamente ao espaço de nomes subjacente.

O tipo é aproximadamente equivalente ao seguinte código:

```
class SimpleNamespace:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __repr__(self):
        keys = sorted(self.__dict__)
        items = ("{}={!r}".format(k, self.__dict__[k]) for k in keys)
        return "{}({})".format(type(self).__name__, ", ".join(items))
```

(continua na próxima página)

(continuação da página anterior)

```
    def __eq__(self, other):
        return self.__dict__ == other.__dict__
```

`SimpleNamespace` pode ser útil como um substituto para `class NS: pass`. No entanto, para um tipo de registro estruturado, use *`namedtuple()`*.

Novo na versão 3.3.

`types.`**`DynamicClassAttribute`**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)
Roteia o acesso ao atributo em uma classe para __getattr__.

Este é um descritor, usado para definir atributos que atuam de forma diferente quando acessados por meio de uma instância e por meio de uma classe. O acesso à instância permanece normal, mas o acesso a um atributo por meio de uma classe será roteado para o método __getattr__ da classe; isso é feito levantando AttributeError.

This allows one to have properties active on an instance, and have virtual attributes on the class with the same name (see Enum for an example).

Novo na versão 3.4.

8.9.4 Funções de utilidade de corrotina

`types.`**`coroutine`**(*gen_func*)
Esta função transforma uma função *geradora* em uma função de corrotina que retorna uma corrotina baseada em gerador. A corrotina baseada em gerador ainda é um *iterador gerador*, mas também é considerada um objeto corrotina e é *aguardável*. No entanto, pode não necessariamente implementar o método `__await__()`.

Se *gen_func* for uma função geradora, ela será modificada no local.

Se *gen_func* não for uma função geradora, ela será envolta. Se ele retornar uma instância de `Collections.abc.Generator`, a instância será envolvida em um objeto proxy *aguardável*. Todos os outros tipos de objetos serão retornados como estão.

Novo na versão 3.5.

8.10 `copy` — Operações de cópia profunda e cópia sombra

Código Fonte: Lib/copy.py

As instruções de atribuição no Python não copiam objetos, elas criam ligações entre um destino e um objeto. Para coleções que são mutáveis ou contêm itens mutáveis, às vezes é necessária uma cópia para que seja possível alterar uma cópia sem alterar a outra. Este módulo fornece operações genéricas de cópia profunda e rasa (explicadas abaixo).

Sumário da Interface:

`copy.`**`copy`**(*x*)
Devolve uma cópia rasa de *x*.

`copy.`**`deepcopy`**(*x*[, *memo*])
Retorna uma cópia profunda de *x*.

`exception` `copy.`**`error`**
Levantada para erros específicos do módulo.

A diferença entre cópia profunda e rasa é relevante apenas para objetos compostos (objetos que contêm outros objetos, como listas ou instâncias de classe):

- Uma *cópia rasa* constrói um novo objeto composto e então (na medida do possível) insere nele *referências* aos objetos encontrados no original.
- Uma *cópia profunda* constrói um novo objeto composto e então, recursivamente, insere nele *cópias* dos objetos encontrados no original.

Frequentemente, existem dois problemas com operações de cópia profunda que não existem com operações de cópia rasa:

- Objetos recursivos (objetos compostos que, direta ou indiretamente, contêm uma referência a si mesmos) podem causar um laço recursivo.
- Como a cópia profunda copia tudo, ela pode copiar muito, como dados que devem ser compartilhados entre as cópias.

A função *deepcopy()* evita esses problemas:

- mantendo um dicionário `memo` de objetos já copiados durante a passagem de cópia atual; e
- permitindo que as classes definidas pelo usuário substituam a operação de cópia ou o conjunto de componentes copiados.

Este módulo não copia tipos como módulo, método, rastreamento de pilha, quadro de pilha, arquivo, soquete, janela, matriz ou outros tipos semelhantes. Ele "copia" funções e classes (rasas e profundas), devolvendo o objeto original inalterado; isso é compatível com a maneira que estes itens são tratados pelo módulo *pickle*.

Cópias rasas de dicionários podem ser feitas usando *dict.copy()*, e de listas atribuindo uma fatia de toda a lista, por exemplo, `lista_copiada = lista_original[:]`.

As classes podem usar as mesmas interfaces para controlar a cópia que usam para controlar o *pickling*. Veja a descrição do módulo *pickle* para informações sobre esses métodos. Na verdade, o módulo *copy* usa as funções pickle registradas do módulo *copyreg*.

Para que uma classe defina sua própria implementação de cópia, ela pode definir métodos especiais `__copy__()` e `__deepcopy__()`. O primeiro é chamado para implementar a operação de cópia rasa; nenhum argumento adicional é passado. O último é chamado para implementar a operação de cópia profunda; é passado um argumento, o dicionário `memo`. Se a implementação de `__deepcopy__()` precisa fazer uma cópia profunda de um componente, ela deve chamar a função *deepcopy()* com o componente como primeiro argumento e o dicionário de memorando como segundo argumento.

Ver também:

Módulo *pickle* Discussão dos métodos especiais usados para dar suporte à recuperação e restauração do estado do objeto.

8.11 `pprint` — Impressão Bonita de Dados

Código Fonte: Lib/pprint.py

The *pprint* module provides a capability to "pretty-print" arbitrary Python data structures in a form which can be used as input to the interpreter. If the formatted structures include objects which are not fundamental Python types, the representation may not be loadable. This may be the case if objects such as files, sockets or classes are included, as well as many other objects which are not representable as Python literals.

The formatted representation keeps objects on a single line if it can, and breaks them onto multiple lines if they don't fit within the allowed width. Construct *PrettyPrinter* objects explicitly if you need to adjust the width constraint.

Dictionaries are sorted by key before the display is computed.

The *pprint* module defines one class:

class pprint.**PrettyPrinter**(*indent=1*, *width=80*, *depth=None*, *stream=None*, *, *compact=False*)

Construct a *PrettyPrinter* instance. This constructor understands several keyword parameters. An output stream may be set using the *stream* keyword; the only method used on the stream object is the file protocol's `write()` method. If not specified, the *PrettyPrinter* adopts `sys.stdout`. The amount of indentation added for each recursive level is specified by *indent*; the default is one. Other values can cause output to look a little odd, but can make nesting easier to spot. The number of levels which may be printed is controlled by *depth*; if the data structure being printed is too deep, the next contained level is replaced by `...`. By default, there is no constraint on the depth of the objects being formatted. The desired output width is constrained using the *width* parameter; the default is 80 characters. If a structure cannot be formatted within the constrained width, a best effort will be made. If *compact* is false (the default) each item of a long sequence will be formatted on a separate line. If *compact* is true, as many items as will fit within the *width* will be formatted on each output line.

Alterado na versão 3.4: Added the *compact* parameter.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[   ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
    'spam',
    'eggs',
    'lumberjack',
    'knights',
    'ni']
>>> pp = pprint.PrettyPrinter(width=41, compact=True)
>>> pp.pprint(stuff)
[['spam', 'eggs', 'lumberjack',
  'knights', 'ni'],
 'spam', 'eggs', 'lumberjack', 'knights',
 'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',)))))))))
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', (...)))))))
```

The *pprint* module also provides several shortcut functions:

pprint.**pformat**(*object*, *indent=1*, *width=80*, *depth=None*, *, *compact=False*)

Return the formatted representation of *object* as a string. *indent*, *width*, *depth* and *compact* will be passed to the *PrettyPrinter* constructor as formatting parameters.

Alterado na versão 3.4: Added the *compact* parameter.

pprint.**pprint**(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, *, *compact=False*)

Prints the formatted representation of *object* on *stream*, followed by a newline. If *stream* is `None`, `sys.stdout` is used. This may be used in the interactive interpreter instead of the *print()* function for inspecting values (you can even reassign `print = pprint.pprint` for use within a scope). *indent*, *width*, *depth* and *compact* will be passed to the *PrettyPrinter* constructor as formatting parameters.

Alterado na versão 3.4: Added the *compact* parameter.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=...>,
```

(continua na próxima página)

(continuação da página anterior)

```
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

pprint.**isreadable**(*object*)

Determine if the formatted representation of *object* is "readable", or can be used to reconstruct the value using *eval()*. This always returns `False` for recursive objects.

```
>>> pprint.isreadable(stuff)
False
```

pprint.**isrecursive**(*object*)

Determine if *object* requires a recursive representation.

One more support function is also defined:

pprint.**saferepr**(*object*)

Return a string representation of *object*, protected against recursive data structures. If the representation of *object* exposes a recursive entry, the recursive reference will be represented as `<Recursion on typename with id=number>`. The representation is not otherwise formatted.

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights', 'ni']"
```

8.11.1 PrettyPrinter Objects

PrettyPrinter instances have the following methods:

PrettyPrinter.**pformat**(*object*)

Return the formatted representation of *object*. This takes into account the options passed to the *PrettyPrinter* constructor.

PrettyPrinter.**pprint**(*object*)

Print the formatted representation of *object* on the configured stream, followed by a newline.

The following methods provide the implementations for the corresponding functions of the same names. Using these methods on an instance is slightly more efficient since new *PrettyPrinter* objects don't need to be created.

PrettyPrinter.**isreadable**(*object*)

Determine if the formatted representation of the object is "readable," or can be used to reconstruct the value using *eval()*. Note that this returns `False` for recursive objects. If the *depth* parameter of the *PrettyPrinter* is set and the object is deeper than allowed, this returns `False`.

PrettyPrinter.**isrecursive**(*object*)

Determine if the object requires a recursive representation.

This method is provided as a hook to allow subclasses to modify the way objects are converted to strings. The default implementation uses the internals of the *saferepr()* implementation.

PrettyPrinter.**format**(*object*, *context*, *maxlevels*, *level*)

Returns three values: the formatted version of *object* as a string, a flag indicating whether the result is readable, and a flag indicating whether recursion was detected. The first argument is the object to be presented. The second is a dictionary which contains the *id()* of objects that are part of the current presentation context (direct and indirect containers for *object* that are affecting the presentation) as the keys; if an object needs to be presented which is already represented in *context*, the third return value should be `True`. Recursive calls to the *format()* method

should add additional entries for containers to this dictionary. The third argument, *maxlevels*, gives the requested limit to recursion; this will be `0` if there is no requested limit. This argument should be passed unmodified to recursive calls. The fourth argument, *level*, gives the current level; recursive calls should be passed a value less than that of the current call.

8.11.2 Exemplo

To demonstrate several uses of the *pprint()* function and its parameters, let's fetch information about a project from PyPI:

```
>>> import json
>>> import pprint
>>> from urllib.request import urlopen
>>> with urlopen('https://pypi.org/pypi/sampleproject/json') as resp:
...     project_info = json.load(resp)['info']
```

In its basic form, *pprint()* shows the whole object:

```
>>> pprint.pprint(project_info)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': ['Development Status :: 3 - Alpha',
                 'Intended Audience :: Developers',
                 'License :: OSI Approved :: MIT License',
                 'Programming Language :: Python :: 2',
                 'Programming Language :: Python :: 2.6',
                 'Programming Language :: Python :: 2.7',
                 'Programming Language :: Python :: 3',
                 'Programming Language :: Python :: 3.2',
                 'Programming Language :: Python :: 3.3',
                 'Programming Language :: Python :: 3.4',
                 'Topic :: Software Development :: Build Tools'],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {'last_day': -1, 'last_month': -1, 'last_week': -1},
 'home_page': 'https://github.com/pypa/sampleproject',
```

(continua na próxima página)

(continuação da página anterior)

```
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {'Download': 'UNKNOWN',
                  'Homepage': 'https://github.com/pypa/sampleproject'},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

The result can be limited to a certain *depth* (ellipsis is used for deeper contents):

```
>>> pprint.pprint(project_info, depth=1)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
```

(continua na próxima página)

(continuação da página anterior)

```
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

Additionally, maximum character *width* can be suggested. If a long object cannot be split, the specified width will be exceeded:

```
>>> pprint.pprint(project_info, depth=1, width=60)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the '
                'project.\n'
                '\n'
                'The file should use UTF-8 encoding and be '
                'written using ReStructured Text. It\n'
                'will be used to generate the project '
                'webpage on PyPI, and should be written '
                'for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would '
                'include an overview of the project, '
                'basic\n'
                'usage examples, etc. Generally, including '
                'the project changelog in here is not\n'
                'a good idea, although a simple "What\'s '
                'New" section for the most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

8.12 reprlib — Alternate repr() implementation

Código Fonte: Lib/reprlib.py

The *reprlib* module provides a means for producing object representations with limits on the size of the resulting strings. This is used in the Python debugger and may be useful in other contexts as well.

This module provides a class, an instance, and a function:

class reprlib.**Repr**
Class which provides formatting services useful in implementing functions similar to the built-in *repr()*; size limits for different object types are added to avoid the generation of representations which are excessively long.

reprlib.**aRepr**
This is an instance of *Repr* which is used to provide the *repr()* function described below. Changing the attributes of this object will affect the size limits used by *repr()* and the Python debugger.

reprlib.**repr**(*obj*)
This is the *repr()* method of `aRepr`. It returns a string similar to that returned by the built-in function of the same name, but with limits on most sizes.

In addition to size-limiting tools, the module also provides a decorator for detecting recursive calls to `__repr__()` and substituting a placeholder string instead.

@reprlib.**recursive_repr**(*fillvalue="..."*)
Decorator for `__repr__()` methods to detect recursive calls within the same thread. If a recursive call is made, the *fillvalue* is returned, otherwise, the usual `__repr__()` call is made. For example:

```
>>> from reprlib import recursive_repr
>>> class MyList(list):
...     @recursive_repr()
...     def __repr__(self):
...         return '<' + '|'.join(map(repr, self)) + '>'
...
>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

Novo na versão 3.2.

8.12.1 Objetos Repr

Repr instances provide several attributes which can be used to provide size limits for the representations of different object types, and methods which format specific object types.

Repr.**maxlevel**
Depth limit on the creation of recursive representations. The default is `6`.

Repr.**maxdict**
Repr.**maxlist**
Repr.**maxtuple**
Repr.**maxset**
Repr.**maxfrozenset**
Repr.**maxdeque**

Repr.**maxarray**
Limits on the number of entries represented for the named object type. The default is 4 for *maxdict*, 5 for *maxarray*, and 6 for the others.

Repr.**maxlong**
Maximum number of characters in the representation for an integer. Digits are dropped from the middle. The default is 40.

Repr.**maxstring**
Limit on the number of characters in the representation of the string. Note that the "normal" representation of the string is used as the character source: if escape sequences are needed in the representation, these may be mangled when the representation is shortened. The default is 30.

Repr.**maxother**
This limit is used to control the size of object types for which no specific formatting method is available on the *Repr* object. It is applied in a similar manner as *maxstring*. The default is 20.

Repr.**repr**(*obj*)
The equivalent to the built-in *repr()* that uses the formatting imposed by the instance.

Repr.**repr1**(*obj*, *level*)
Recursive implementation used by *repr()*. This uses the type of *obj* to determine which formatting method to call, passing it *obj* and *level*. The type-specific methods should call *repr1()* to perform recursive formatting, with `level - 1` for the value of *level* in the recursive call.

Repr.**repr_TYPE**(*obj*, *level*)
Formatting methods for specific types are implemented as methods with a name based on the type name. In the method name, **TYPE** is replaced by `'_'.join(type(obj).__name__.split())`. Dispatch to these methods is handled by *repr1()*. Type-specific methods which need to recursively format a value should call `self.repr1(subobj, level - 1)`.

8.12.2 Subclassing Repr Objects

The use of dynamic dispatching by *Repr.repr1()* allows subclasses of *Repr* to add support for additional built-in object types or to modify the handling of types already supported. This example shows how special support for file objects could be added:

```
import reprlib
import sys

class MyRepr(reprlib.Repr):

    def repr_TextIOWrapper(self, obj, level):
        if obj.name in {'<stdin>', '<stdout>', '<stderr>'}:
            return obj.name
        return repr(obj)

aRepr = MyRepr()
print(aRepr.repr(sys.stdin))          # prints '<stdin>'
```

8.13 enum — Suporte a enumerações

Novo na versão 3.4.

Código Fonte: Lib/enum.py

Uma enumeração é um conjunto de nomes simbólicos (membros) vinculados a valores únicos e constantes. Dentro de uma enumeração, os membros podem ser comparados por identidade, e a enumeração em si pode ser iterada.

8.13.1 Conteúdo do Módulo

Este módulo define quatro classes de enumeração que podem ser usadas para definir conjuntos de nomes e valores: *Enum*, *IntEnum*, *Flag*, and *IntFlag*. Ele também define um decorator, *unique()*, e um auxiliar, *auto*.

class enum.**Enum**
: Classe base para criação de constantes enumeradas. Veja a seção *API Funcional* para uma sintaxe alternativa de construção.

class enum.**IntEnum**
: Base class for creating enumerated constants that are also subclasses of *int*.

class enum.**IntFlag**
: Base class for creating enumerated constants that can be combined using the bitwise operators without losing their *IntFlag* membership. *IntFlag* members are also subclasses of *int*.

class enum.**Flag**
: Base class for creating enumerated constants that can be combined using the bitwise operations without losing their *Flag* membership.

enum.**unique**()
: Enum class decorator that ensures only one name is bound to any one value.

class enum.**auto**
: Instances are replaced with an appropriate value for Enum members. Initial value starts at 1.

Novo na versão 3.6: `Flag`, `IntFlag`, `auto`

8.13.2 Creating an Enum

Enumerations are created using the `class` syntax, which makes them easy to read and write. An alternative creation method is described in *Functional API*. To define an enumeration, subclass *Enum* as follows:

```
>>> from enum import Enum
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...
```

Nota: Enum member values

Member values can be anything: *int*, *str*, etc.. If the exact value is unimportant you may use *auto* instances and an appropriate value will be chosen for you. Care must be taken if you mix *auto* with other values.

Nota: Nomenclature

- The class `Color` is an *enumeration* (or *enum*)
- The attributes `Color.RED`, `Color.GREEN`, etc., are *enumeration members* (or *enum members*) and are functionally constants.
- The enum members have *names* and *values* (the name of `Color.RED` is `RED`, the value of `Color.BLUE` is `3`, etc.)

Nota: Even though we use the `class` syntax to create Enums, Enums are not normal Python classes. See *How are Enums different?* for more details.

Enumeration members have human readable string representations:

```
>>> print(Color.RED)
Color.RED
```

…while their `repr` has more information:

```
>>> print(repr(Color.RED))
<Color.RED: 1>
```

The *type* of an enumeration member is the enumeration it belongs to:

```
>>> type(Color.RED)
<enum 'Color'>
>>> isinstance(Color.GREEN, Color)
True
>>>
```

Enum members also have a property that contains just their item name:

```
>>> print(Color.RED.name)
RED
```

Enumerations support iteration, in definition order:

```
>>> class Shake(Enum):
...     VANILLA = 7
...     CHOCOLATE = 4
...     COOKIES = 9
...     MINT = 3
...
>>> for shake in Shake:
...     print(shake)
...
Shake.VANILLA
Shake.CHOCOLATE
Shake.COOKIES
Shake.MINT
```

Enumeration members are hashable, so they can be used in dictionaries and sets:

```
>>> apples = {}
>>> apples[Color.RED] = 'red delicious'
>>> apples[Color.GREEN] = 'granny smith'
>>> apples == {Color.RED: 'red delicious', Color.GREEN: 'granny smith'}
True
```

8.13.3 Programmatic access to enumeration members and their attributes

Sometimes it's useful to access members in enumerations programmatically (i.e. situations where `Color.RED` won't do because the exact color is not known at program-writing time). `Enum` allows such access:

```
>>> Color(1)
<Color.RED: 1>
>>> Color(3)
<Color.BLUE: 3>
```

If you want to access enum members by *name*, use item access:

```
>>> Color['RED']
<Color.RED: 1>
>>> Color['GREEN']
<Color.GREEN: 2>
```

If you have an enum member and need its `name` or `value`:

```
>>> member = Color.RED
>>> member.name
'RED'
>>> member.value
1
```

8.13.4 Duplicating enum members and values

Having two enum members with the same name is invalid:

```
>>> class Shape(Enum):
...     SQUARE = 2
...     SQUARE = 3
...
Traceback (most recent call last):
...
TypeError: Attempted to reuse key: 'SQUARE'
```

However, two enum members are allowed to have the same value. Given two members A and B with the same value (and A defined first), B is an alias to A. By-value lookup of the value of A and B will return A. By-name lookup of B will also return A:

```
>>> class Shape(Enum):
...     SQUARE = 2
...     DIAMOND = 1
...     CIRCLE = 3
...     ALIAS_FOR_SQUARE = 2
...
```

(continua na próxima página)

(continuação da página anterior)

```
>>> Shape.SQUARE
<Shape.SQUARE: 2>
>>> Shape.ALIAS_FOR_SQUARE
<Shape.SQUARE: 2>
>>> Shape(2)
<Shape.SQUARE: 2>
```

Nota: Attempting to create a member with the same name as an already defined attribute (another member, a method, etc.) or attempting to create an attribute with the same name as a member is not allowed.

8.13.5 Ensuring unique enumeration values

By default, enumerations allow multiple names as aliases for the same value. When this behavior isn't desired, the following decorator can be used to ensure each value is used only once in the enumeration:

`@enum.unique`

A `class` decorator specifically for enumerations. It searches an enumeration's `__members__` gathering any aliases it finds; if any are found *ValueError* is raised with the details:

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

8.13.6 Usando valores automáticos

If the exact value is unimportant you can use *auto*:

```
>>> from enum import Enum, auto
>>> class Color(Enum):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

The values are chosen by `_generate_next_value_()`, which can be overridden:

```
>>> class AutoName(Enum):
...     def _generate_next_value_(name, start, count, last_values):
...         return name
...
>>> class Ordinal(AutoName):
```

(continua na próxima página)

(continuação da página anterior)

```
...     NORTH = auto()
...     SOUTH = auto()
...     EAST = auto()
...     WEST = auto()
...
>>> list(Ordinal)
[<Ordinal.NORTH: 'NORTH'>, <Ordinal.SOUTH: 'SOUTH'>, <Ordinal.EAST: 'EAST'>, <Ordinal.
↪WEST: 'WEST'>]
```

Nota: The goal of the default `_generate_next_value_()` methods is to provide the next *int* in sequence with the last *int* provided, but the way it does this is an implementation detail and may change.

Nota: The `_generate_next_value_()` method must be defined before any members.

8.13.7 Iteração

Iterating over the members of an enum does not provide the aliases:

```
>>> list(Shape)
[<Shape.SQUARE: 2>, <Shape.DIAMOND: 1>, <Shape.CIRCLE: 3>]
```

The special attribute `__members__` is an ordered dictionary mapping names to members. It includes all names defined in the enumeration, including the aliases:

```
>>> for name, member in Shape.__members__.items():
...     name, member
...
('SQUARE', <Shape.SQUARE: 2>)
('DIAMOND', <Shape.DIAMOND: 1>)
('CIRCLE', <Shape.CIRCLE: 3>)
('ALIAS_FOR_SQUARE', <Shape.SQUARE: 2>)
```

The `__members__` attribute can be used for detailed programmatic access to the enumeration members. For example, finding all the aliases:

```
>>> [name for name, member in Shape.__members__.items() if member.name != name]
['ALIAS_FOR_SQUARE']
```

8.13.8 Comparações

Enumeration members are compared by identity:

```
>>> Color.RED is Color.RED
True
>>> Color.RED is Color.BLUE
False
>>> Color.RED is not Color.BLUE
True
```

Ordered comparisons between enumeration values are *not* supported. Enum members are not integers (but see *IntEnum* below):

```
>>> Color.RED < Color.BLUE
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'Color' and 'Color'
```

Equality comparisons are defined though:

```
>>> Color.BLUE == Color.RED
False
>>> Color.BLUE != Color.RED
True
>>> Color.BLUE == Color.BLUE
True
```

Comparisons against non-enumeration values will always compare not equal (again, *IntEnum* was explicitly designed to behave differently, see below):

```
>>> Color.BLUE == 2
False
```

8.13.9 Allowed members and attributes of enumerations

The examples above use integers for enumeration values. Using integers is short and handy (and provided by default by the *Functional API*), but not strictly enforced. In the vast majority of use-cases, one doesn't care what the actual value of an enumeration is. But if the value *is* important, enumerations can have arbitrary values.

Enumerations are Python classes, and can have methods and special methods as usual. If we have this enumeration:

```
>>> class Mood(Enum):
...     FUNKY = 1
...     HAPPY = 3
...
...     def describe(self):
...         # self is the member here
...         return self.name, self.value
...
...     def __str__(self):
...         return 'my custom str! {0}'.format(self.value)
...
...     @classmethod
...     def favorite_mood(cls):
...         # cls here is the enumeration
...         return cls.HAPPY
...
```

Then:

```
>>> Mood.favorite_mood()
<Mood.HAPPY: 3>
>>> Mood.HAPPY.describe()
('HAPPY', 3)
>>> str(Mood.FUNKY)
'my custom str! 1'
```

The rules for what is allowed are as follows: names that start and end with a single underscore are reserved by enum and cannot be used; all other attributes defined within an enumeration will become members of this enumeration, with the exception of special methods (`__str__()`, `__add__()`, etc.), descriptors (methods are also descriptors), and variable names listed in `_ignore_`.

Note: if your enumeration defines `__new__()` and/or `__init__()` then whatever value(s) were given to the enum member will be passed into those methods. See *Planet* for an example.

8.13.10 Restricted Enum subclassing

A new *Enum* class must have one base Enum class, up to one concrete data type, and as many *object*-based mixin classes as needed. The order of these base classes is:

```
class EnumName([mix-in, ...,] [data-type,] base-enum):
    pass
```

Also, subclassing an enumeration is allowed only if the enumeration does not define any members. So this is forbidden:

```
>>> class MoreColor(Color):
...     PINK = 17
...
Traceback (most recent call last):
...
TypeError: Cannot extend enumerations
```

But this is allowed:

```
>>> class Foo(Enum):
...     def some_behavior(self):
...         pass
...
>>> class Bar(Foo):
...     HAPPY = 1
...     SAD = 2
...
```

Allowing subclassing of enums that define members would lead to a violation of some important invariants of types and instances. On the other hand, it makes sense to allow sharing some common behavior between a group of enumerations. (See *OrderedEnum* for an example.)

8.13.11 Pickling

Enumerations can be pickled and unpickled:

```
>>> from test.test_enum import Fruit
>>> from pickle import dumps, loads
>>> Fruit.TOMATO is loads(dumps(Fruit.TOMATO))
True
```

The usual restrictions for pickling apply: picklable enums must be defined in the top level of a module, since unpickling requires them to be importable from that module.

Nota: With pickle protocol version 4 it is possible to easily pickle enums nested in other classes.

It is possible to modify how Enum members are pickled/unpickled by defining *__reduce_ex__()* in the enumeration class.

8.13.12 API funcional

The *Enum* class is callable, providing the following functional API:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG')
>>> Animal
<enum 'Animal'>
>>> Animal.ANT
<Animal.ANT: 1>
>>> Animal.ANT.value
1
>>> list(Animal)
[<Animal.ANT: 1>, <Animal.BEE: 2>, <Animal.CAT: 3>, <Animal.DOG: 4>]
```

The semantics of this API resemble *namedtuple*. The first argument of the call to *Enum* is the name of the enumeration.

The second argument is the *source* of enumeration member names. It can be a whitespace-separated string of names, a sequence of names, a sequence of 2-tuples with key/value pairs, or a mapping (e.g. dictionary) of names to values. The last two options enable assigning arbitrary values to enumerations; the others auto-assign increasing integers starting with 1 (use the `start` parameter to specify a different starting value). A new class derived from *Enum* is returned. In other words, the above assignment to `Animal` is equivalent to:

```
>>> class Animal(Enum):
...     ANT = 1
...     BEE = 2
...     CAT = 3
...     DOG = 4
...
```

The reason for defaulting to `1` as the starting number and not `0` is that `0` is `False` in a boolean sense, but enum members all evaluate to `True`.

Pickling enums created with the functional API can be tricky as frame stack implementation details are used to try and figure out which module the enumeration is being created in (e.g. it will fail if you use a utility function in separate module, and also may not work on IronPython or Jython). The solution is to specify the module name explicitly as follows:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', module=__name__)
```

Aviso: If `module` is not supplied, and Enum cannot determine what it is, the new Enum members will not be unpicklable; to keep errors closer to the source, pickling will be disabled.

The new pickle protocol 4 also, in some circumstances, relies on *__qualname__* being set to the location where pickle will be able to find the class. For example, if the class was made available in class SomeData in the global scope:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', qualname='SomeData.Animal')
```

The complete signature is:

```
Enum(value='NewEnumName', names=<...>, *, module='...', qualname='...', type=<mixed-
↪in class>, start=1)
```

value What the new Enum class will record as its name.

nomes The Enum members. This can be a whitespace or comma separated string (values will start at 1 unless otherwise specified):

```
'RED GREEN BLUE' | 'RED,GREEN,BLUE' | 'RED, GREEN, BLUE'
```

or an iterator of names:

```
['RED', 'GREEN', 'BLUE']
```

or an iterator of (name, value) pairs:

```
[('CYAN', 4), ('MAGENTA', 5), ('YELLOW', 6)]
```

or a mapping:

```
{'CHARTREUSE': 7, 'SEA_GREEN': 11, 'ROSEMARY': 42}
```

módulo name of module where new Enum class can be found.

qualname where in module new Enum class can be found.

tipo type to mix in to new Enum class.

start number to start counting at if only names are passed in.

Alterado na versão 3.5: The *start* parameter was added.

8.13.13 Derived Enumerations

IntEnum

The first variation of *Enum* that is provided is also a subclass of *int*. Members of an *IntEnum* can be compared to integers; by extension, integer enumerations of different types can also be compared to each other:

```
>>> from enum import IntEnum
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Request(IntEnum):
...     POST = 1
...     GET = 2
...
>>> Shape == 1
False
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE == Request.POST
True
```

However, they still can't be compared to standard *Enum* enumerations:

```
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
```

(continua na próxima página)

(continuação da página anterior)

```
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...
>>> Shape.CIRCLE == Color.RED
False
```

IntEnum values behave like integers in other ways you'd expect:

```
>>> int(Shape.CIRCLE)
1
>>> ['a', 'b', 'c'][Shape.CIRCLE]
'b'
>>> [i for i in range(Shape.SQUARE)]
[0, 1]
```

IntFlag

The next variation of *Enum* provided, *IntFlag*, is also based on *int*. The difference being *IntFlag* members can be combined using the bitwise operators (&, |, ^, ~) and the result is still an *IntFlag* member. However, as the name implies, *IntFlag* members also subclass *int* and can be used wherever an *int* is used. Any operation on an *IntFlag* member besides the bit-wise operations will lose the *IntFlag* membership.

Novo na versão 3.6.

Sample *IntFlag* class:

```
>>> from enum import IntFlag
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...
>>> Perm.R | Perm.W
<Perm.R|W: 6>
>>> Perm.R + Perm.W
6
>>> RW = Perm.R | Perm.W
>>> Perm.R in RW
True
```

It is also possible to name the combinations:

```
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...     RWX = 7
>>> Perm.RWX
<Perm.RWX: 7>
>>> ~Perm.RWX
<Perm.-8: -8>
```

Another important difference between *IntFlag* and *Enum* is that if no flags are set (the value is 0), its boolean evaluation is *False*:

```
>>> Perm.R & Perm.X
<Perm.0: 0>
>>> bool(Perm.R & Perm.X)
False
```

Because *IntFlag* members are also subclasses of *int* they can be combined with them:

```
>>> Perm.X | 8
<Perm.8|X: 9>
```

Flag

The last variation is *Flag*. Like *IntFlag*, *Flag* members can be combined using the bitwise operators (&, |, ^, ~). Unlike *IntFlag*, they cannot be combined with, nor compared against, any other *Flag* enumeration, nor *int*. While it is possible to specify the values directly it is recommended to use *auto* as the value and let *Flag* select an appropriate value.

Novo na versão 3.6.

Like *IntFlag*, if a combination of *Flag* members results in no flags being set, the boolean evaluation is *False*:

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.RED & Color.GREEN
<Color.0: 0>
>>> bool(Color.RED & Color.GREEN)
False
```

Individual flags should have values that are powers of two (1, 2, 4, 8, …), while combinations of flags won't:

```
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...     WHITE = RED | BLUE | GREEN
...
>>> Color.WHITE
<Color.WHITE: 7>
```

Giving a name to the "no flags set" condition does not change its boolean value:

```
>>> class Color(Flag):
...     BLACK = 0
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.BLACK
<Color.BLACK: 0>
>>> bool(Color.BLACK)
False
```

Nota: For the majority of new code, *Enum* and *Flag* are strongly recommended, since *IntEnum* and *IntFlag* break some semantic promises of an enumeration (by being comparable to integers, and thus by transitivity to other unrelated enumerations). *IntEnum* and *IntFlag* should be used only in cases where *Enum* and *Flag* will not do; for example, when integer constants are replaced with enumerations, or for interoperability with other systems.

Others

While *IntEnum* is part of the *enum* module, it would be very simple to implement independently:

```
class IntEnum(int, Enum):
    pass
```

This demonstrates how similar derived enumerations can be defined; for example a `StrEnum` that mixes in *str* instead of *int*.

Some rules:

1. When subclassing *Enum*, mix-in types must appear before *Enum* itself in the sequence of bases, as in the *IntEnum* example above.
2. While *Enum* can have members of any type, once you mix in an additional type, all the members must have values of that type, e.g. *int* above. This restriction does not apply to mix-ins which only add methods and don't specify another data type such as *int* or *str*.
3. When another data type is mixed in, the `value` attribute is *not the same* as the enum member itself, although it is equivalent and will compare equal.
4. %-style formatting: *%s* and *%r* call the *Enum* class's `__str__()` and `__repr__()` respectively; other codes (such as *%i* or *%h* for IntEnum) treat the enum member as its mixed-in type.
5. Formatted string literals, *str.format()*, and *format()* will use the mixed-in type's `__format__()`. If the *Enum* class's *str()* or *repr()* is desired, use the *!s* or *!r* format codes.

8.13.14 Interesting examples

While *Enum*, *IntEnum*, *IntFlag*, and *Flag* are expected to cover the majority of use-cases, they cannot cover them all. Here are recipes for some different types of enumerations that can be used directly, or as examples for creating one's own.

Omitting values

In many use-cases one doesn't care what the actual value of an enumeration is. There are several ways to define this type of simple enumeration:

- use instances of *auto* for the value
- use instances of *object* as the value
- use a descriptive string as the value
- use a tuple as the value and a custom `__new__()` to replace the tuple with an *int* value

Using any of these methods signifies to the user that these values are not important, and also enables one to add, remove, or reorder members without having to renumber the remaining members.

Whichever method you choose, you should provide a *repr()* that also hides the (unimportant) value:

```
>>> class NoValue(Enum):
...     def __repr__(self):
...         return '<%s.%s>' % (self.__class__.__name__, self.name)
...
```

Using `auto`

Using *`auto`* would look like:

```
>>> class Color(NoValue):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.GREEN
<Color.GREEN>
```

Using `object`

Using *`object`* would look like:

```
>>> class Color(NoValue):
...     RED = object()
...     GREEN = object()
...     BLUE = object()
...
>>> Color.GREEN
<Color.GREEN>
```

Using a descriptive string

Using a string as the value would look like:

```
>>> class Color(NoValue):
...     RED = 'stop'
...     GREEN = 'go'
...     BLUE = 'too fast!'
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
'go'
```

Usando um `__new__()` personalizado

Using an auto-numbering `__new__()` would look like:

```
>>> class AutoNumber(NoValue):
...     def __new__(cls):
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
>>> class Color(AutoNumber):
...     RED = ()
...     GREEN = ()
...     BLUE = ()
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
2
```

Nota: The `__new__()` method, if defined, is used during creation of the Enum members; it is then replaced by Enum's `__new__()` which is used after class creation for lookup of existing members.

OrderedEnum

An ordered enumeration that is not based on *IntEnum* and so maintains the normal *Enum* invariants (such as not being comparable to other enumerations):

```
>>> class OrderedEnum(Enum):
...     def __ge__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value >= other.value
...         return NotImplemented
...     def __gt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value > other.value
...         return NotImplemented
...     def __le__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value <= other.value
...         return NotImplemented
...     def __lt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value < other.value
...         return NotImplemented
...
>>> class Grade(OrderedEnum):
...     A = 5
...     B = 4
...     C = 3
...     D = 2
...     F = 1
...
```

(continua na próxima página)

(continuação da página anterior)

```
>>> Grade.C < Grade.A
True
```

DuplicateFreeEnum

Raises an error if a duplicate member name is found instead of creating an alias:

```
>>> class DuplicateFreeEnum(Enum):
...     def __init__(self, *args):
...         cls = self.__class__
...         if any(self.value == e.value for e in cls):
...             a = self.name
...             e = cls(self.value).name
...             raise ValueError(
...                 "aliases not allowed in DuplicateFreeEnum:  %r --> %r"
...                 % (a, e))
...
>>> class Color(DuplicateFreeEnum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     GRENE = 2
...
Traceback (most recent call last):
...
ValueError: aliases not allowed in DuplicateFreeEnum:  'GRENE' --> 'GREEN'
```

Nota: This is a useful example for subclassing Enum to add or change other behaviors as well as disallowing aliases. If the only desired change is disallowing aliases, the *unique()* decorator can be used instead.

Planet

If `__new__()` or `__init__()` is defined the value of the enum member will be passed to those methods:

```
>>> class Planet(Enum):
...     MERCURY = (3.303e+23, 2.4397e6)
...     VENUS   = (4.869e+24, 6.0518e6)
...     EARTH   = (5.976e+24, 6.37814e6)
...     MARS    = (6.421e+23, 3.3972e6)
...     JUPITER = (1.9e+27,   7.1492e7)
...     SATURN  = (5.688e+26, 6.0268e7)
...     URANUS  = (8.686e+25, 2.5559e7)
...     NEPTUNE = (1.024e+26, 2.4746e7)
...     def __init__(self, mass, radius):
...         self.mass = mass       # in kilograms
...         self.radius = radius   # in meters
...     @property
...     def surface_gravity(self):
...         # universal gravitational constant  (m3 kg-1 s-2)
...         G = 6.67300E-11
...         return G * self.mass / (self.radius * self.radius)
```

(continua na próxima página)

(continuação da página anterior)

```
...
>>> Planet.EARTH.value
(5.976e+24, 6378140.0)
>>> Planet.EARTH.surface_gravity
9.802652743337129
```

TimePeriod

An example to show the _ignore_ attribute in use:

```
>>> from datetime import timedelta
>>> class Period(timedelta, Enum):
...     "different lengths of time"
...     _ignore_ = 'Period i'
...     Period = vars()
...     for i in range(367):
...         Period['day_%d' % i] = i
...
>>> list(Period)[:2]
[<Period.day_0: datetime.timedelta(0)>, <Period.day_1: datetime.timedelta(days=1)>]
>>> list(Period)[-2:]
[<Period.day_365: datetime.timedelta(days=365)>, <Period.day_366: datetime.
↪timedelta(days=366)>]
```

8.13.15 How are Enums different?

Enums have a custom metaclass that affects many aspects of both derived Enum classes and their instances (members).

Enum Classes

The `EnumMeta` metaclass is responsible for providing the `__contains__()`, `__dir__()`, `__iter__()` and other methods that allow one to do things with an *Enum* class that fail on a typical class, such as *list(Color)* or *some_enum_var in Color*. `EnumMeta` is responsible for ensuring that various other methods on the final *Enum* class are correct (such as `__new__()`, *__getnewargs__()*, `__str__()` and `__repr__()`).

Enum Members (aka instances)

The most interesting thing about Enum members is that they are singletons. `EnumMeta` creates them all while it is creating the *Enum* class itself, and then puts a custom `__new__()` in place to ensure that no new ones are ever instantiated by returning only the existing member instances.

Finer Points

Supported __dunder__ names

`__members__` is an `OrderedDict` of `member_name:member` items. It is only available on the class.

`__new__()`, if specified, must create and return the enum members; it is also a very good idea to set the member's `_value_` appropriately. Once all the members are created it is no longer used.

Supported _sunder_ names

- `_name_` – name of the member
- `_value_` – value of the member; can be set / modified in `__new__`
- `_missing_` – a lookup function used when a value is not found; may be overridden
- `_ignore_` – a list of names, either as a *`list()`* or a *`str()`*, that will not be transformed into members, and will be removed from the final class
- `_order_` – used in Python 2/3 code to ensure member order is consistent (class attribute, removed during class creation)
- `_generate_next_value_` – used by the *Functional API* and by `auto` to get an appropriate value for an enum member; may be overridden

Novo na versão 3.6: `_missing_`, `_order_`, `_generate_next_value_`

Novo na versão 3.7: `_ignore_`

To help keep Python 2 / Python 3 code in sync an `_order_` attribute can be provided. It will be checked against the actual order of the enumeration and raise an error if the two do not match:

```
>>> class Color(Enum):
...     _order_ = 'RED GREEN BLUE'
...     RED = 1
...     BLUE = 3
...     GREEN = 2
...
Traceback (most recent call last):
...
TypeError: member order does not match _order_
```

Nota: In Python 2 code the `_order_` attribute is necessary as definition order is lost before it can be recorded.

Enum member type

`Enum` members are instances of their *`Enum`* class, and are normally accessed as `EnumClass.member`. Under certain circumstances they can also be accessed as `EnumClass.member.member`, but you should never do this as that lookup may fail or, worse, return something besides the *`Enum`* member you are looking for (this is another good reason to use all-uppercase names for members):

```
>>> class FieldTypes(Enum):
...     name = 0
...     value = 1
...     size = 2
...
>>> FieldTypes.value.size
<FieldTypes.size: 2>
>>> FieldTypes.size.value
2
```

Alterado na versão 3.5.

Boolean value of `Enum` classes and members

Enum members that are mixed with non-*Enum* types (such as *int*, *str*, etc.) are evaluated according to the mixed-in type's rules; otherwise, all members evaluate as *True*. To make your own Enum's boolean evaluation depend on the member's value add the following to your class:

```
def __bool__(self):
    return bool(self.value)
```

Enum classes always evaluate as *True*.

`Enum` classes with methods

If you give your *Enum* subclass extra methods, like the *Planet* class above, those methods will show up in a *dir()* of the member, but not of the class:

```
>>> dir(Planet)
['EARTH', 'JUPITER', 'MARS', 'MERCURY', 'NEPTUNE', 'SATURN', 'URANUS', 'VENUS', '__
↪class__', '__doc__', '__members__', '__module__']
>>> dir(Planet.EARTH)
['__class__', '__doc__', '__module__', 'name', 'surface_gravity', 'value']
```

Combining members of `Flag`

If a combination of Flag members is not named, the *repr()* will include all named flags and all named combinations of flags that are in the value:

```
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...     MAGENTA = RED | BLUE
...     YELLOW = RED | GREEN
...     CYAN = GREEN | BLUE
...
>>> Color(3)  # named combination
<Color.YELLOW: 3>
>>> Color(7)      # not named combination
<Color.CYAN|MAGENTA|BLUE|YELLOW|GREEN|RED: 7>
```

CAPÍTULO 9

Módulos Matemáticos e Numéricos

Os módulos descritos neste capítulo fornecem funções e tipos de dados relacionados com o numérico e matemática. O módulo: mod: *numbers* define uma hierarquia abstrata de tipos numéricos. Os módulos: mod: *math* e: mod:‘ cmath‘ contêm várias funções matemáticas para números de ponto flutuante e números complexos. O módulo: mod: *decimal* suporta representações exatas de números decimais, usando aritmética de precisão arbitrária.

Os seguintes módulos estão documentados neste capítulo:

9.1 `numbers` — Classes base abstratas numéricas

Código Fonte: Lib/numbers.py

The *numbers* module (**PEP 3141**) defines a hierarchy of numeric *abstract base classes* which progressively define more operations. None of the types defined in this module can be instantiated.

class numbers.**Number**
A raiz da hierarquia numérica. Se você quiser apenas verificar se um argumento *x* é um número, sem se importar com o tipo, use `isinstance(x, Number)`.

9.1.1 A torre numérica

class numbers.**Complex**
Subclasses of this type describe complex numbers and include the operations that work on the built-in *complex* type. These are: conversions to *complex* and *bool*, *real*, *imag*, +, −, *, /, *abs()*, *conjugate()*, ==, and !=. All except − and != are abstract.

real
Abstrata. Obtém o componente real deste número.

imag
Abstrata. Obtém o componente imaginário deste número.

abstractmethod conjugate()
Abstrata. Retorna o conjugado complexo. Por exemplo, `(1+3j).conjugate() == (1-3j)`.

class numbers.**Real**
Para *Complex*, *Real* adiciona as operações que funcionam em números reais.

Em suma, são: uma conversão para *float*, *math.trunc()*, *round()*, *math.floor()*, *math.ceil()*, *divmod()*, //, %, <, <=, > e >=.

Real também fornece padrão para *complex()*, *real*, *imag* e *conjugate()*.

class numbers.**Rational**
Estende *Real* e adiciona as propriedades *numerator* e *denominator*, que devem estar nos termos mais baixos. Com eles, ele fornece um padrão para *float()*.

numerator
Abstrato.

denominator
Abstrato.

class numbers.**Integral**
Subtypes *Rational* and adds a conversion to *int*. Provides defaults for *float()*, *numerator*, and *denominator*. Adds abstract methods for ** and bit-string operations: <<, >>, &, ^, |, ~.

9.1.2 Nota para implementadores de tipos

Os implementadores devem ter o cuidado de tornar iguais números iguais e fazer hash deles com os mesmos valores. Isso pode ser sutil se houver duas extensões diferentes dos números reais. Por exemplo, *fractions.Fraction* implementa *hash()* desta forma:

```
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on
        # simple fractions.
        return hash((self.numerator, self.denominator))
```

Adicionando mais ABCs numéricas

Existem, é claro, mais ABCs possíveis para números, e isso seria uma hierarquia pobre se excluísse a possibilidade de adicioná-los. Você pode adicionar `MyFoo` entre *Complex* e *Real* com:

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

Implementando as operações aritméticas

Queremos implementar as operações aritméticas de forma que as operações de modo misto chamem uma implementação cujo autor conhecia os tipos de ambos os argumentos ou convertam ambos para o tipo embutido mais próximo e façam a operação lá. Para subtipos de *Integral*, isso significa que `__add__()` e `__radd__()` devem ser definidos com:

```
class MyIntegral(Integral):

    def __add__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

Existem 5 casos diferentes para uma operação de tipo misto em subclasses de *Complex*. Vou me referir a todo o código acima que não se refere a `MyIntegral` e `OtherTypeIKnowAbout` com um "modelo". `a` será uma instância de `A`, que é um subtipo de *Complex* (`a : A <: Complex`) e `b : B <: Complex`. Vou considerar `a + b`:

1. Se `A` define um `__add__()`, que aceita `b`, tudo está bem.
2. Se `A` voltar ao código padrão e tivesse que retornar um valor de `__add__()`, perderíamos a possibilidade de que `B` definisse um `__radd__()` mais inteligente, então o código padrão deve retornar *NotImplemented* de `__add__()`. (Ou `A` pode não implementar `__add__()`.)
3. Então, `__radd__()` do `B` consegue uma chance. Se ele aceitar `a`, está tudo bem.
4. Se ele recorrer ao padrão, não há mais métodos possíveis para tentar, então é aqui que a implementação padrão deve residir.
5. Se `B <: A`, Python tenta `B.__radd__` antes de `A.__add__`. Isso está ok, porque foi implementado com conhecimento de `A`, então ele pode lidar com essas instâncias antes de delegar para *Complex*.

Se `A <: Complex` e `B <: Real` sem compartilhar nenhum outro conhecimento, então a operação compartilhada apropriada é aquela envolvendo a *complex* embutida, e ambos `__radd__()` s chegam lá, de forma que `a+b == b+a`.

Como a maioria das operações em qualquer tipo será muito semelhante, pode ser útil definir uma função auxiliar que gera as instâncias de avanço e reversão de qualquer operador. Por exemplo, *fractions.Fraction* usa:

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, Fraction)):
            return monomorphic_operator(a, b)
```

(continua na próxima página)

(continuação da página anterior)

```
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
            return monomorphic_operator(a, b)
        elif isinstance(a, numbers.Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, numbers.Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
    reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse

def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...
```

9.2 `math` — Funções matemáticas

Este módulo fornece acesso às funções matemáticas definidas pelo padrão C.

Essas funções não podem ser usadas com números complexos; use as funções de mesmo nome do módulo *`cmath`* se você precisar de suporte para números complexos. A distinção entre funções que suportam números complexos e aquelas que não suportam é feita uma vez que a maioria dos usuários não quer aprender a matemática necessária para entender números complexos. Receber uma exceção em vez de um resultado complexo permite a detecção antecipada do número complexo inesperado usado como parâmetro, para que o programador possa determinar como e por que ele foi gerado em primeiro lugar.

As funções a seguir são fornecidas por este módulo. Exceto quando explicitamente indicado de outra forma, todos os valores de retorno são pontos flutuantes.

9.2.1 Funções teóricas dos números e de representação

`math.`**`ceil`**(*x*)
Retorna o teto de *x*, o menor inteiro maior ou igual a *x*. Se *x* não for um ponto flutuante, delega para `x.__ceil__()`, que deve retornar um valor de *Integral*.

`math.`**`copysign`**(*x*, *y*)
Retorna um ponto flutuante com a magnitude (valor absoluto) de *x*, mas o sinal de *y*. Em plataformas que suportam zeros sem sinal, `copysign(1.0, -0.0)` retorna *-1.0*.

`math.`**`fabs`**(*x*)
Retorna o valor absoluto de *x*.

`math.`**`factorial`**(*x*)
Retorna *x* fatorial como um inteiro. Levanta *ValueError* se *x* não for integral ou for negativo.

`math.`**`floor`**(*x*)
Retorna o piso de *x*, o maior inteiro menor ou igual a *x*. Se *x* não for um ponto flutuante, delega para `x.__floor__()`, que deve retornar um valor de *Integral*.

`math.`**`fmod`**(*x*, *y*)
Retorna `fmod(x, y)`, conforme definido pela biblioteca C da plataforma. Observe que a expressão Python `x % y` pode não retornar o mesmo resultado. A intenção do padrão C é que `fmod(x, y)` seja exatamente (matematicamente; com precisão infinita) igual a `x - n*y` para algum inteiro *n* de modo que o resultado tenha o mesmo sinal que *x* e magnitude menor que `abs(y)`. O `x % y` do Python retorna um resultado com o sinal de *y*, e pode não ser exatamente computável para argumentos de ponto flutuante. Por exemplo, `fmod(-1e-100, 1e100)` é `-1e-100`, mas o resultado de `-1e-100 % 1e100` do Python é `1e100-1e-100`, que não pode ser representado exatamente como um ponto flutuante, e é arredondado para o surpreendente `1e100`. Por esta razão, a função *fmod()* é geralmente preferida ao trabalhar com pontos flutuantes, enquanto o `x % y` do Python é preferido ao trabalhar com inteiros.

`math.`**`frexp`**(*x*)
Retorna a mantissa e o expoente de *x* como o par `(m, e)`. *m* é um ponto flutuante e *e* é um inteiro tal que `x == m * 2**e` exatamente. Se *x* for zero, retorna `(0.0, 0)`, caso contrário, `0.5 <= abs(m) < 1`. Isso é usado para "separar" a representação interna de um ponto flutuante de forma portátil.

`math.`**`fsum`**(*iterable*)
Retorna uma soma de valores de ponto flutuante precisa no iterável. Evita perda de precisão rastreando várias somas parciais intermediárias:

```
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
1.0
```

A precisão do algoritmo depende das garantias aritméticas IEEE-754 e do caso típico em que o modo de arredondamento é meio par. Em algumas compilações que não são do Windows, a biblioteca C subjacente usa adição de precisão estendida e pode ocasionalmente arredondar uma soma intermediária fazendo com que ela fique fora do bit menos significativo.

Para uma discussão mais aprofundada e duas abordagens alternativas, consulte as receitas do livro de receitas ASPN para soma de ponto flutuante preciso.

`math.`**`gcd`**(*a*, *b*)
Return the greatest common divisor of the integers *a* and *b*. If either *a* or *b* is nonzero, then the value of `gcd(a, b)` is the largest positive integer that divides both *a* and *b*. `gcd(0, 0)` returns `0`.

Novo na versão 3.5.

math.**isclose**(*a*, *b*, *, *rel_tol=1e-09*, *abs_tol=0.0*)
Retorna `True` se os valores *a* e *b* estiverem próximos e `False` caso contrário.

Se dois valores são ou não considerados próximos, é determinado de acordo com as tolerâncias absolutas e relativas fornecidas.

rel_tol é a tolerância relativa – é a diferença máxima permitida entre *a* e *b*, em relação ao maior valor absoluto de *a* e *b*. Por exemplo, para definir uma tolerância de 5%, passe `rel_tol=0.05`. A tolerância padrão é `1e-09`, o que garante que os dois valores sejam iguais em cerca de 9 dígitos decimais. *rel_tol* deve ser maior que zero.

abs_tol é a tolerância absoluta mínima – útil para comparações próximas a zero. *abs_tol* deve ser pelo menos zero.

Se nenhum erro ocorrer, o resultado será: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

Os valores especiais do IEEE 754 de `NaN`, `inf` e `-inf` serão tratados de acordo com as regras do IEEE. Especificamente, `NaN` não é considerado próximo a qualquer outro valor, incluindo `NaN`. `inf` e `-inf` são considerados apenas próximos a si mesmos.

Novo na versão 3.5.

Ver também:

PEP 485 – Uma função para testar igualdade aproximada

math.**isfinite**(*x*)
Retorna `True` se *x* não for um infinito nem um NaN, e `False` caso contrário. (Observe que `0.0` *é* considerado finito.)

Novo na versão 3.2.

math.**isinf**(*x*)
Retorna `True` se *x* for um infinito positivo ou negativo, e `False` caso contrário.

math.**isnan**(*x*)
Retorna `True` se *x* for um NaN (não um número), e `False` caso contrário.

math.**ldexp**(*x*, *i*)
Retorna `x * (2**i)`. Este é essencialmente o inverso da função *frexp()*.

math.**modf**(*x*)
Retorna as partes fracionárias e inteiras de *x*. Ambos os resultados carregam o sinal de *x* e são pontos flutuantes.

math.**remainder**(*x*, *y*)
Retorna o resto no estilo IEEE 754 de *x* em relação a *y*. Para o finito *x* e o finito diferente de zero *y*, esta é a diferença `x - n*y`, onde `n` é o número inteiro mais próximo do valor exato do quociente `x / y`. Se `x / y` está exatamente no meio do caminho entre dois inteiros consecutivos, o inteiro *even* mais próximo é usado para `n`. O resto `r = remainder(x, y)` assim sempre satisfaz `abs(r) <= 0.5 * abs(y)`.

Casos especiais seguem IEEE 754: em particular, `remainder(x, math.inf)` é *x* para qualquer *x* finito, e `remainder(x, 0)` e `remainder(math.inf, x)` levantam *ValueError* para qualquer *x* não NaN. Se o resultado da operação `remainder` for zero, esse zero terá o mesmo sinal de *x*.

Em plataformas que usam ponto flutuante binário do IEEE 754, o resultado dessa operação é sempre exatamente representável: nenhum erro de arredondamento é introduzido.

Novo na versão 3.7.

math.**trunc**(*x*)
Retorna o valor *x* *Real* truncado com um *Integral* (geralmente um inteiro). Delega para `x.__trunc__()`.

Observe que *frexp()* e *modf()* têm um padrão de chamada/retorno diferente de seus equivalentes C: elas pegam um único argumento e retornam um par de valores, ao invés de retornar seu segundo valor de retorno por meio de um "parâmetro de saída" (não existe tal coisa em Python).

Para as funções *ceil()*, *floor()* e *modf()*, observe que *todos* os números de ponto flutuante de magnitude suficientemente grande são inteiros exatos. Os pontos flutuantes do Python normalmente não carregam mais do que 53 bits de precisão (o mesmo que o tipo duplo da plataforma C), caso em que qualquer flutuante *x* com `abs(x) >= 2**52` necessariamente não tem bits fracionários.

9.2.2 Funções de potência e logarítmicas

`math.`**`exp`**(*x*)
Retorna *e* elevado à potência *x*, onde *e* = 2.718281… é a base dos logaritmos naturais. Isso geralmente é mais preciso do que `math.e ** x` ou `pow(math.e, x)`.

`math.`**`expm1`**(*x*)
Retorna *e* elevado à potência *x*, menos 1. Aqui *e* é a base dos logaritmos naturais. Para pequenos pontos flutuantes *x*, a subtração em `exp(x) - 1` pode resultar em uma perda significativa de precisão; a função *expm1()* fornece uma maneira de calcular essa quantidade com precisão total:

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

Novo na versão 3.2.

`math.`**`log`**(*x*[, *base*])
Com um argumento, retorna o logaritmo natural de *x* (para base *e*).

Com dois argumentos, retorna o logaritmo de *x* para a *base* fornecida, calculada como `log(x)/log(base)`.

`math.`**`log1p`**(*x*)
Retorna o logaritmo natural de *1+x* (base *e*). O resultado é calculado de forma precisa para *x* próximo a zero.

`math.`**`log2`**(*x*)
Retorna o logaritmo de base 2 de *x*. Isso geralmente é mais preciso do que `log(x, 2)`.

Novo na versão 3.3.

Ver também:

int.bit_length() retorna o número de bits necessários para representar um inteiro em binário, excluindo o sinal e os zeros à esquerda.

`math.`**`log10`**(*x*)
Retorna o logaritmo de base 10 de *x*. Isso geralmente é mais preciso do que `log(x, 10)`.

`math.`**`pow`**(*x*, *y*)
Retorna `x` elevado à potência `y`. Os casos excepcionais seguem o Anexo 'F' da norma C99, tanto quanto possível. Em particular, `pow(1.0, x)` e `pow(x, 0.0)` sempre retornam `1.0`, mesmo quando `x` é um zero ou um NaN. Se ambos `x` e `y` são finitos, `x` é negativo, e `y` não é um inteiro, então `pow(x, y)` é indefinido e levanta *ValueError*.

Ao contrário do operador embutido `**`, *math.pow()* converte ambos os seus argumentos para o tipo *float*. Use `**` ou a função embutida *pow()* para calcular potências inteiras exatas.

`math.`**`sqrt`**(*x*)
Retorna a raiz quadrada de *x*.

9.2.3 Funções trigonométricas

math.**acos**(*x*)
Return the arc cosine of *x*, in radians.

math.**asin**(*x*)
Return the arc sine of *x*, in radians.

math.**atan**(*x*)
Return the arc tangent of *x*, in radians.

math.**atan2**(*y*, *x*)
Retorna `atan(y / x)`, em radianos. O resultado está entre `-pi` e `pi`. O vetor no plano da origem ao ponto `(x, y)` faz este ângulo com o eixo X positivo. O ponto de *atan2()* é que os sinais de ambas as entradas são conhecidos por ele, então ele pode calcular o quadrante correto para o ângulo. Por exemplo, `atan(1)` e `atan2(1, 1)` são ambos"pi/4", mas `atan2(-1, -1)` é `-3*pi/4`.

math.**cos**(*x*)
Retorna o cosseno de *x* radianos.

math.**hypot**(*x*, *y*)
Return the Euclidean norm, `sqrt(x*x + y*y)`. This is the length of the vector from the origin to point `(x, y)`.

math.**sin**(*x*)
Retorna o seno de *x* radianos.

math.**tan**(*x*)
Retorna o tangente de *x* radianos.

9.2.4 Conversão angular

math.**degrees**(*x*)
Converte o ângulo *x* de radianos para graus.

math.**radians**(*x*)
Converte o ângulo *x* de graus para radianos.

9.2.5 Funções hiperbólicas

Funções hiperbólicas são análogos de funções trigonométricas baseadas em hipérboles em vez de círculos.

math.**acosh**(*x*)
Retorna o cosseno hiperbólico inverso de *x*.

math.**asinh**(*x*)
Retorna o seno hiperbólico inverso de *x*.

math.**atanh**(*x*)
Retorna a tangente hiperbólica inversa de *x*.

math.**cosh**(*x*)
Retorna o cosseno hiperbólico de *x*.

math.**sinh**(*x*)
Retorna o seno hiperbólico de *x*.

math.**tanh**(*x*)
Retorna a tangente hiperbólica de *x*.

9.2.6 Funções especiais

math.**erf**(*x*)
Retorna a função erro em *x*.

A função *erf()* pode ser usada para calcular funções estatísticas tradicionais, como a distribuição normal padrão cumulativa:

```
def phi(x):
    'Cumulative distribution function for the standard normal distribution'
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

Novo na versão 3.2.

math.**erfc**(*x*)
Retorna a função erro complementar em *x*. A função erro complementar é definida como `1.0 - erf(x)`. É usado para grandes valores de *x* onde uma subtração de um causaria uma perda de significância.

Novo na versão 3.2.

math.**gamma**(*x*)
Retorna a função gama em *x*.

Novo na versão 3.2.

math.**lgamma**(*x*)
Retorna o logaritmo natural do valor absoluto da função gama em *x*.

Novo na versão 3.2.

9.2.7 Constantes

math.**pi**
A constante matemática $\pi = 3.141592\ldots$, para a precisão disponível.

math.**e**
A constante matemática $e = 2.718281\ldots$, para a precisão disponível.

math.**tau**
A constante matemática $\tau = 6.283185\ldots$, para a precisão disponível. Tau é uma constante de círculo igual a 2π, a razão entre a circunferência de um círculo e seu raio. Para saber mais sobre Tau, confira o vídeo Pi is (still) Wrong de Vi Hart, e comece a comemorar o dia do Tau comendo duas vezes mais torta!

Novo na versão 3.6.

math.**inf**
Um infinito positivo de ponto flutuante. (Para infinito negativo, use `-math.inf`.) Equivalente à saída de `float('inf')`.

Novo na versão 3.5.

math.**nan**
Um valor de ponto flutuante "não um número" (NaN). Equivalente à saída de `float('nan')`.

Novo na versão 3.5.

CPython implementation detail: O módulo *math* consiste principalmente em O módulo: mod: *math* consiste principalmente em invólucros finos em torno das funções da biblioteca matemática C da plataforma. O comportamento em casos excepcionais segue o Anexo F da norma C99 quando apropriado. A implementação atual levantará *ValueError* para operações inválidas como `sqrt(-1.0)` ou `log(0.0)` (onde C99 Anexo F recomenda sinalizar operação inválida ou divisão por zero), e *OverflowError* para resultados que estouram (por exemplo,"exp(1000.0)"). Um NaN

não será retornado de nenhuma das funções acima, a menos que um ou mais dos argumentos de entrada sejam um NaN; nesse caso, a maioria das funções retornará um NaN, mas (novamente seguindo C99 Anexo F) há algumas exceções a esta regra, por exemplo, `pow(float('nan'), 0.0)` ou `hypot(float('nan'), float('inf'))`.

Observe que o Python não faz nenhum esforço para distinguir NaNs de sinalização de NaNs silenciosos, e o comportamento para NaNs de sinalização permanece não especificado. O comportamento típico é tratar todos os NaNs como se estivessem quietos.

Ver também:

Modulo *cmath* Versões de números complexos de muitas dessas funções.

9.3 cmath — Funções matemáticas para números complexos

This module provides access to mathematical functions for complex numbers. The functions in this module accept integers, floating-point numbers or complex numbers as arguments. They will also accept any Python object that has either a `__complex__()` or a `__float__()` method: these methods are used to convert the object to a complex or floating-point number, respectively, and the function is then applied to the result of the conversion.

Nota: On platforms with hardware and system-level support for signed zeros, functions involving branch cuts are continuous on *both* sides of the branch cut: the sign of the zero distinguishes one side of the branch cut from the other. On platforms that do not support signed zeros the continuity is as specified below.

9.3.1 Conversions to and from polar coordinates

A Python complex number `z` is stored internally using *rectangular* or *Cartesian* coordinates. It is completely determined by its *real part* `z.real` and its *imaginary part* `z.imag`. In other words:

```
z == z.real + z.imag*1j
```

Polar coordinates give an alternative way to represent a complex number. In polar coordinates, a complex number *z* is defined by the modulus *r* and the phase angle *phi*. The modulus *r* is the distance from *z* to the origin, while the phase *phi* is the counterclockwise angle, measured in radians, from the positive x-axis to the line segment that joins the origin to *z*.

The following functions can be used to convert from the native rectangular coordinates to polar coordinates and back.

`cmath.`**`phase`**`(x)`

Return the phase of *x* (also known as the *argument* of *x*), as a float. `phase(x)` is equivalent to `math.atan2(x.imag, x.real)`. The result lies in the range [$-\pi$, π], and the branch cut for this operation lies along the negative real axis, continuous from above. On systems with support for signed zeros (which includes most systems in current use), this means that the sign of the result is the same as the sign of `x.imag`, even when `x.imag` is zero:

```
>>> phase(complex(-1.0, 0.0))
3.141592653589793
>>> phase(complex(-1.0, -0.0))
-3.141592653589793
```

Nota: The modulus (absolute value) of a complex number *x* can be computed using the built-in *abs()* function. There is no separate *cmath* module function for this operation.

cmath.**polar**(*x*)
Return the representation of *x* in polar coordinates. Returns a pair `(r, phi)` where *r* is the modulus of *x* and phi is the phase of *x*. `polar(x)` is equivalent to `(abs(x), phase(x))`.

cmath.**rect**(*r*, *phi*)
Return the complex number *x* with polar coordinates *r* and *phi*. Equivalent to `r * (math.cos(phi) + math.sin(phi)*1j)`.

9.3.2 Funções de potência e logarítmicas

cmath.**exp**(*x*)
Return *e* raised to the power *x*, where *e* is the base of natural logarithms.

cmath.**log**(*x*[, *base*])
Returns the logarithm of *x* to the given *base*. If the *base* is not specified, returns the natural logarithm of *x*. There is one branch cut, from 0 along the negative real axis to -∞, continuous from above.

cmath.**log10**(*x*)
Return the base-10 logarithm of *x*. This has the same branch cut as *log()*.

cmath.**sqrt**(*x*)
Return the square root of *x*. This has the same branch cut as *log()*.

9.3.3 Funções trigonométricas

cmath.**acos**(*x*)
Return the arc cosine of *x*. There are two branch cuts: One extends right from 1 along the real axis to ∞, continuous from below. The other extends left from -1 along the real axis to -∞, continuous from above.

cmath.**asin**(*x*)
Return the arc sine of *x*. This has the same branch cuts as *acos()*.

cmath.**atan**(*x*)
Return the arc tangent of *x*. There are two branch cuts: One extends from `1j` along the imaginary axis to ∞`j`, continuous from the right. The other extends from `-1j` along the imaginary axis to −∞`j`, continuous from the left.

cmath.**cos**(*x*)
Return the cosine of *x*.

cmath.**sin**(*x*)
Devolve o seno de *x*.

cmath.**tan**(*x*)
Return the tangent of *x*.

9.3.4 Funções hiperbólicas

cmath.**acosh**(*x*)
Return the inverse hyperbolic cosine of *x*. There is one branch cut, extending left from 1 along the real axis to -∞, continuous from above.

cmath.**asinh**(*x*)
Return the inverse hyperbolic sine of *x*. There are two branch cuts: One extends from `1j` along the imaginary axis to ∞`j`, continuous from the right. The other extends from `-1j` along the imaginary axis to −∞`j`, continuous from the left.

cmath.**atanh**(*x*)
Return the inverse hyperbolic tangent of *x*. There are two branch cuts: One extends from `1` along the real axis to ∞, continuous from below. The other extends from `-1` along the real axis to -∞, continuous from above.

cmath.**cosh**(*x*)
Retorna o cosseno hiperbólico de *x*.

cmath.**sinh**(*x*)
Retorna o seno hiperbólico de *x*.

cmath.**tanh**(*x*)
Retorna a tangente hiperbólica de *x*.

9.3.5 Classification functions

cmath.**isfinite**(*x*)
Return `True` if both the real and imaginary parts of *x* are finite, and `False` otherwise.

Novo na versão 3.2.

cmath.**isinf**(*x*)
Return `True` if either the real or the imaginary part of *x* is an infinity, and `False` otherwise.

cmath.**isnan**(*x*)
Return `True` if either the real or the imaginary part of *x* is a NaN, and `False` otherwise.

cmath.**isclose**(*a*, *b*, ***, *rel_tol=1e-09*, *abs_tol=0.0*)
Retorna `True` se os valores *a* e *b* estiverem próximos e `False` caso contrário.

Se dois valores são ou não considerados próximos, é determinado de acordo com as tolerâncias absolutas e relativas fornecidas.

rel_tol é a tolerância relativa – é a diferença máxima permitida entre *a* e *b*, em relação ao maior valor absoluto de *a* e *b*. Por exemplo, para definir uma tolerância de 5%, passe `rel_tol=0.05`. A tolerância padrão é `1e-09`, o que garante que os dois valores sejam iguais em cerca de 9 dígitos decimais. *rel_tol* deve ser maior que zero.

abs_tol é a tolerância absoluta mínima – útil para comparações próximas a zero. *abs_tol* deve ser pelo menos zero.

Se nenhum erro ocorrer, o resultado será: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

Os valores especiais do IEEE 754 de `NaN`, `inf` e `-inf` serão tratados de acordo com as regras do IEEE. Especificamente, `NaN` não é considerado próximo a qualquer outro valor, incluindo `NaN`. `inf` e `-inf` são considerados apenas próximos a si mesmos.

Novo na versão 3.5.

Ver também:

PEP 485 – Uma função para testar igualdade aproximada

9.3.6 Constantes

cmath.**pi**
The mathematical constant π, as a float.

cmath.**e**
The mathematical constant e, as a float.

cmath.**tau**
The mathematical constant τ, as a float.

Novo na versão 3.6.

cmath.**inf**
Floating-point positive infinity. Equivalent to `float('inf')`.

Novo na versão 3.6.

cmath.**infj**
Complex number with zero real part and positive infinity imaginary part. Equivalent to `complex(0.0, float('inf'))`.

Novo na versão 3.6.

cmath.**nan**
A floating-point "not a number" (NaN) value. Equivalent to `float('nan')`.

Novo na versão 3.6.

cmath.**nanj**
Complex number with zero real part and NaN imaginary part. Equivalent to `complex(0.0, float('nan'))`.

Novo na versão 3.6.

Note that the selection of functions is similar, but not identical, to that in module *math*. The reason for having two modules is that some users aren't interested in complex numbers, and perhaps don't even know what they are. They would rather have `math.sqrt(-1)` raise an exception than return a complex number. Also note that the functions defined in *cmath* always return a complex number, even if the answer can be expressed as a real number (in which case the complex number has an imaginary part of zero).

A note on branch cuts: They are curves along which the given function fails to be continuous. They are a necessary feature of many complex functions. It is assumed that if you need to compute with complex functions, you will understand about branch cuts. Consult almost any (not too elementary) book on complex variables for enlightenment. For information of the proper choice of branch cuts for numerical purposes, a good reference should be the following:

Ver também:

Kahan, W: Branch cuts for complex elementary functions; or, Much ado about nothing's sign bit. In Iserles, A., and Powell, M. (eds.), The state of the art in numerical analysis. Clarendon Press (1987) pp165–211.

9.4 `decimal` — Aritmética de ponto decimal fixo e ponto flutuante

Código Fonte: Lib/decimal.py

O módulo *`decimal`* fornece suporte a aritmética rápida de ponto flutuante decimal corretamente arredondado. Oferece várias vantagens sobre o tipo de dados *`float`*:

- Decimal "é baseado em um modelo de ponto flutuante que foi projetado com as pessoas em mente e necessariamente tem um princípio orientador primordial – os computadores devem fornecer uma aritmética que funcione da mesma maneira que a aritmética que as pessoas aprendem na escola". – trecho da especificação aritmética decimal.
- Os números decimais podem ser representados exatamente. Por outro lado, números como `1.1` e `2.2` não possuem representações exatas em ponto flutuante binário. Os usuários finais normalmente não esperam que `1.1 + 2.2` sejam exibidos como `3.3000000000000003`, como acontece com o ponto flutuante binário.
- A exatidão transita para a aritmética. No ponto flutuante decimal, `0.1 + 0.1 + 0.1 - 0.3` é exatamente igual a zero. No ponto flutuante binário, o resultado é `5.5511151231257827e-017`. Embora próximas de zero, as diferenças impedem o teste de igualdade confiável e as diferenças podem se acumular. Por esse motivo, o decimal é preferido em aplicativos de contabilidade que possuem invariáveis estritos de igualdade.
- O módulo decimal incorpora uma noção de casas significativas para que `1.30 + 1.20` seja `2.50`. O zero à direita é mantido para indicar significância. Esta é a apresentação habitual para aplicações monetárias. Para multiplicação, a abordagem "livro escolar" usa todas as figuras nos multiplicandos. Por exemplo, `1.3 * 1.2` é igual a `1.56` enquanto `1.30 * 1.20` é igual a `1.5600`.
- Diferentemente do ponto flutuante binário baseado em hardware, o módulo decimal possui uma precisão alterável pelo usuário (padrão de 28 casas), que pode ser tão grande quanto necessário para um determinado problema:

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- O ponto flutuante binário e decimal é implementado em termos de padrões publicados. Enquanto o tipo ponto flutuante embutido expõe apenas uma parte modesta de seus recursos, o módulo decimal expõe todas as partes necessárias do padrão. Quando necessário, o programador tem controle total sobre o arredondamento e o manuseio do sinal. Isso inclui uma opção para impor aritmética exata usando exceções para bloquear quaisquer operações inexatas.
- O módulo decimal foi projetado para dar suporte, "sem prejuízo, a aritmética decimal não arredondada exata (às vezes chamada aritmética de ponto fixo) e aritmética arredondada de ponto flutuante". – trecho da especificação aritmética decimal.

O design do módulo é centrado em torno de três conceitos: o número decimal, o contexto da aritmética e os sinais.

Um número decimal é imutável. Possui um sinal, dígitos de coeficiente e um expoente. Para preservar a significância, os dígitos do coeficiente não truncam zeros à direita. Os decimais também incluem valores especiais, tais como `Infinity`, `-Infinity` e `NaN`. O padrão também diferencia `-0` de `+0`.

O contexto da aritmética é um ambiente que especifica precisão, regras de arredondamento, limites de expoentes, sinalizadores indicando os resultados das operações e ativadores de interceptação que determinam se os sinais são tratados como exceções. As opções de arredondamento incluem *`ROUND_CEILING`*, *`ROUND_DOWN`*, *`ROUND_FLOOR`*, *`ROUND_HALF_DOWN`*, *`ROUND_HALF_EVEN`*, *`ROUND_HALF_UP`*, *`ROUND_UP`* e *`ROUND_05UP`*.

Sinais são grupos de condições excepcionais que surgem durante o curso da computação. Dependendo das necessidades do aplicativo, os sinais podem ser ignorados, considerados informativos ou tratados como exceções. Os sinais no módulo decimal são: *Clamped*, *InvalidOperation*, *DivisionByZero*, *Inexact*, *Rounded*, *Subnormal*, *Overflow*, *Underflow* e *FloatOperation*.

Para cada sinal, há um sinalizador e um ativador de interceptação. Quando um sinal é encontrado, seu sinalizador é definido como um e, se o ativador de interceptação estiver definido como um, uma exceção será gerada. Os sinalizadores são fixos; portanto, o usuário precisa redefini-los antes de monitorar um cálculo.

Ver também:

- A especificação geral aritmética decimal da IBM, The General Decimal Arithmetic Specification.

9.4.1 Tutorial de início rápido

O início usual do uso de decimais é importar o módulo, exibir o contexto atual com *getcontext()* e, se necessário, definir novos valores para precisão, arredondamento ou armadilhas ativados:

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[Overflow, DivisionByZero,
        InvalidOperation])

>>> getcontext().prec = 7       # Set a new precision
```

Instâncias decimais podem ser construídas a partir de números inteiros, strings, pontos flutuantes ou tuplas. A construção de um número inteiro ou de um ponto flutuante realiza uma conversão exata do valor desse número inteiro ou ponto flutuante. Os números decimais incluem valores especiais como `NaN`, que significa "Não é um número", `Infinity` positivo e negativo e `-0`:

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.4142135623730950488016887242')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

Se o sinal *FloatOperation* for capturado na armadilha, a mistura acidental de decimais e pontos flutuantes em construtores ou comparações de ordenação levanta uma exceção:

```
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(continua na próxima página)

(continuação da página anterior)

```
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') == 3.5
True
```

Novo na versão 3.3.

O significado de um novo decimal é determinado apenas pelo número de dígitos digitados. A precisão e o arredondamento do contexto só entram em jogo durante operações aritméticas.

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

Se os limites internos da versão C forem excedidos, a construção de um decimal levanta *InvalidOperation*:

```
>>> Decimal("1e9999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [<class 'decimal.InvalidOperation'>]
```

Alterado na versão 3.3.

Os decimais interagem bem com grande parte do restante do Python. Aqui está um pequeno circo voador de ponto flutuante decimal:

```
>>> data = list(map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split()))
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)
Decimal('1.3')
>>> int(a)
1
>>> a * 5
Decimal('6.70')
```

(continua na próxima página)

(continuação da página anterior)

```
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```

E algumas funções matemáticas também estão disponíveis no Decimal:

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801688724')
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')
>>> Decimal('10').log10()
Decimal('1')
```

O método `quantize()` arredonda um número para um expoente fixo. Esse método é útil para aplicativos monetários que geralmente arredondam os resultados para um número fixo de locais:

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

Como mostrado acima, a função *getcontext()* acessa o contexto atual e permite que as configurações sejam alteradas. Essa abordagem atende às necessidades da maioria das aplicações.

Para trabalhos mais avançados, pode ser útil criar contextos alternativos usando o construtor Context(). Para ativar uma alternativa, use a função *setcontext()*.

De acordo com o padrão, o módulo *decimal* fornece dois contextos padrão prontos para uso, *BasicContext* e *ExtendedContext*. O primeiro é especialmente útil para depuração porque muitas das armadilhas estão ativadas:

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

Os contextos também possuem sinalizadores para monitorar condições excepcionais encontradas durante os cálculos. Os sinalizadores permanecem definidos até que sejam explicitamente limpos, portanto, é melhor limpar os sinalizadores antes de cada conjunto de cálculos monitorados usando o método `clear_flags()`.

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[Inexact, Rounded], traps=[])
```

A entrada *flags* mostra que a aproximação racional de `Pi` foi arredondada (dígitos além da precisão do contexto foram descartados) e que o resultado é inexato (alguns dos dígitos descartados eram diferentes de zero).

As armadilhas individuais são definidas usando o dicionário no campo `traps` de um contexto:

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

A maioria dos programas ajusta o contexto atual apenas uma vez, no início do programa. E, em muitas aplicações, os dados são convertidos para *`Decimal`* com uma única conversão dentro de um loop. Com o conjunto de contextos e decimais criados, a maior parte do programa manipula os dados de maneira diferente do que com outros tipos numéricos do Python.

9.4.2 Objetos de Decimal

class decimal.**Decimal**(*value="0"*, *context=None*)

Constrói um novo objeto de *`Decimal`* com base em *value*.

value pode ser um inteiro, string, tupla, *`float`* ou outro *`Decimal`*. Se nenhum *value* for fornecido, retornará `Decimal('0')`. Se *value* for uma string, ele deverá estar em conformidade com a sintaxe numérica decimal após caracteres de espaço em branco à esquerda e à direita, bem como sublinhados em toda parte, serem removidos:

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

Outros dígitos decimais Unicode também são permitidos onde `dígito` aparece acima. Isso inclui dígitos decimais de vários outros alfabetos (por exemplo, dígitos em árabe-índico e devanāgarī), além dos dígitos de largura total `'\uff10'` a `'\uff19'`.

Se *value* for uma *`tupla`*, ele deverá ter três componentes, um sinal (`0` para positivo ou `1` para negativo), uma *`tupla`* de dígitos e um expoente inteiro. Por exemplo, `Decimal((0, (1, 4, 1, 4), -3))` returna `Decimal('1.414')`.

Se *value* é um *float*, o valor do ponto flutuante binário é convertido sem perdas no seu equivalente decimal exato. Essa conversão geralmente requer 53 ou mais dígitos de precisão. Por exemplo, `Decimal(float('1.1'))` converte para `Decimal('1.100000000000000088817841970012523233890533447265625')`.

A precisão *context* não afeta quantos dígitos estão armazenados. Isso é determinado exclusivamente pelo número de dígitos em *value*. Por exemplo, `Decimal('3.00000')` registra todos os cinco zeros, mesmo que a precisão do contexto seja apenas três.

O objetivo do argumento *context* é determinar o que fazer se *value* for uma string malformada. Se o contexto capturar *InvalidOperation*, uma exceção será levantada; caso contrário, o construtor retornará um novo decimal com o valor de `NaN`.

Uma vez construídos, objetos de *Decimal* são imutáveis.

Alterado na versão 3.2: O argumento para o construtor agora pode ser uma instância *float*.

Alterado na versão 3.3: Os argumentos de *float* levantam uma exceção se a armadilha *FloatOperation* estiver definida. Por padrão, a armadilha está desativada.

Alterado na versão 3.6: Sublinhados são permitidos para agrupamento, como nos literais de ponto flutuante e integral no código.

Objetos decimais de ponto flutuante compartilham muitas propriedades com outros tipos numéricos internos, como *float* e *int*. Todas as operações matemáticas usuais e métodos especiais se aplicam. Da mesma forma, objetos decimais podem ser copiados, separados, impressos, usados como chaves de dicionário, usados como elementos de conjunto, comparados, classificados e coagidos a outro tipo (como *float* ou *int*).

Existem algumas pequenas diferenças entre aritmética em objetos decimais e aritmética em números inteiros e flutuantes. Quando o operador restante `%` é aplicado a objetos decimais, o sinal do resultado é o sinal do *dividend* em vez do sinal do divisor:

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

O operador de divisão inteira `//` se comporta de maneira análoga, retornando a parte inteira do quociente verdadeiro (truncando em direção a zero) em vez de seu piso, de modo a preservar a identidade usual `x == (x // y) * y + x % y`:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

Os operadores `%` e `//` implementam as operações de `módulo` e `divisão inteira` (respectivamente) como descrito na especificação.

Objetos decimais geralmente não podem ser combinados com pontos flutuantes ou instâncias de *fractions.Fraction* em operações aritméticas: uma tentativa de adicionar um *Decimal* a um *float*, por exemplo, levantará um *TypeError*. No entanto, é possível usar os operadores de comparação do Python para comparar uma instância de *Decimal* `x` com outro número `y`. Isso evita resultados confusos ao fazer comparações de igualdade entre números de tipos diferentes.

Alterado na versão 3.2: As comparações de tipos mistos entre instâncias de *Decimal* e outros tipos numéricos agora são totalmente suportadas.

Além das propriedades numéricas padrão, os objetos de ponto flutuante decimal também possuem vários métodos especializados:

adjusted()
Retorna o expoente ajustado depois de deslocar os dígitos mais à direita do coeficiente até restar apenas o dígito principal: `Decimal('321e+5').adjusted()` retorna sete. Usado para determinar a posição do dígito mais significativo em relação ao ponto decimal.

as_integer_ratio()
Retorna um par `(n, d)` de números inteiros que representam a instância dada *Decimal* como uma fração, nos termos mais baixos e com um denominador positivo:

```
>>> Decimal('-3.14').as_integer_ratio()
(-157, 50)
```

A conversão é exata. Levanta OverflowError em infinitos e ValueError em NaNs.

Novo na versão 3.6.

as_tuple()
Retorna uma representação de *tupla nomeada* do número: `DecimalTuple(sinal, dígitos, exponente)`.

canonical()
Retorna a codificação canônica do argumento. Atualmente, a codificação de uma instância de *Decimal* é sempre canônica, portanto, esta operação retorna seu argumento inalterado.

compare(*other*, *context=None*)
Compara os valores de duas instâncias decimais. *compare()* retorna uma instância decimal, e se qualquer operando for um NaN, o resultado será um NaN:

```
a or b is a NaN  ==> Decimal('NaN')
a < b            ==> Decimal('-1')
a == b           ==> Decimal('0')
a > b            ==> Decimal('1')
```

compare_signal(*other*, *context=None*)
Esta operação é idêntica ao método *compare()*, exceto que todos os NaNs sinalizam. Ou seja, se nenhum operando for um NaN sinalizador, qualquer operando NaN silencioso será tratado como se fosse um NaN sinalizador.

compare_total(*other*, *context=None*)
Compara dois operandos usando sua representação abstrata em vez de seu valor numérico. Semelhante ao método *compare()*, mas o resultado fornece uma ordem total nas instâncias de *Decimal*. Duas instâncias de *Decimal* com o mesmo valor numérico, mas diferentes representações se comparam desiguais nesta ordem:

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

Os NaNs silenciosos e sinalizadores também estão incluídos no pedido total. O resultado dessa função é `Decimal('0')` se os dois operandos tiverem a mesma representação, `Decimal('-1')` se o primeiro operando for menor na ordem total que o segundo e `Decimal('1')` se o primeiro operando for maior na ordem total que o segundo operando. Veja a especificação para detalhes da ordem total.

Esta operação não é afetada pelo contexto e é silenciosa: nenhum sinalizador é alterado e nenhum arredondamento é executado. Como uma exceção, a versão C pode levantar InvalidOperation se o segundo operando não puder ser convertido exatamente.

compare_total_mag(*other*, *context=None*)
Compara dois operandos usando sua representação abstrata em vez de seu valor, como em

compare_total(), mas ignorando o sinal de cada operando. `x.compare_total_mag(y)` é equivalente a `x.copy_abs().compare_total(y.copy_abs())`.

Esta operação não é afetada pelo contexto e é silenciosa: nenhum sinalizador é alterado e nenhum arredondamento é executado. Como uma exceção, a versão C pode levantar InvalidOperation se o segundo operando não puder ser convertido exatamente.

conjugate()
Apenas retorna a si próprio, sendo esse método apenas para atender à Especificação Decimal.

copy_abs()
Retorna o valor absoluto do argumento. Esta operação não é afetada pelo contexto e é silenciosa: nenhum sinalizador é alterado e nenhum arredondamento é executado.

copy_negate()
Retorna a negação do argumento. Esta operação não é afetada pelo contexto e é silenciosa: nenhum sinalizador é alterado e nenhum arredondamento é executado.

copy_sign(*other*, *context=None*)
Retorna uma cópia do primeiro operando com o sinal definido para ser o mesmo que o sinal do segundo operando. Por exemplo:

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

Esta operação não é afetada pelo contexto e é silenciosa: nenhum sinalizador é alterado e nenhum arredondamento é executado. Como uma exceção, a versão C pode levantar InvalidOperation se o segundo operando não puder ser convertido exatamente.

exp(*context=None*)
Retorna o valor da função exponencial (natural) `e**x` no número especificado. O resultado é arredondado corretamente usando o modo de arredondamento *ROUND_HALF_EVEN*.

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

from_float(*f*)
Método de classe que converte um ponto flutuante em um número decimal, exatamente.

Observe que *Decimal.from_float(0.1)* não é o mesmo que *Decimal('0.1')*. Como 0.1 não é exatamente representável no ponto flutuante binário, o valor é armazenado como o valor representável mais próximo que é *0x1.999999999999ap-4*. Esse valor equivalente em decimal é *0.1000000000000000055511151231257827021181583404541015625*.

Nota: A partir do Python 3.2 em diante, uma instância de *Decimal* também pode ser construída diretamente a partir de um *float*.

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

Novo na versão 3.1.

fma(*other*, *third*, *context=None*)
Multiplicação e adição fundidos. Retorna self*other+third sem arredondamento do produto intermediário self*other.

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

is_canonical()
Retorna *True* se o argumento for canônico e *False* caso contrário. Atualmente, uma instância de *Decimal* é sempre canônica, portanto, esta operação sempre retorna *True*.

is_finite()
Retorna *True* se o argumento for um número finito e *False* se o argumento for um infinito ou um NaN.

is_infinite()
Retorna *True* se o argumento for infinito positivo ou negativo e *False* caso contrário.

is_nan()
Retorna *True* se o argumento for NaN (silencioso ou sinalizador) e *False* caso contrário.

is_normal(*context=None*)
Retorna *True* se o argumento for um número finito *normal*. Retorna *False* se o argumento for zero, subnormal, infinito ou NaN.

is_qnan()
Retorna *True* se o argumento for um NaN silencioso, e *False* caso contrário.

is_signed()
Retorna *True* se o argumento tiver um sinal negativo e *False* caso contrário. Observe que zeros e NaNs podem carregar sinais.

is_snan()
Retorna *True* se o argumento for um sinal NaN e *False* caso contrário.

is_subnormal(*context=None*)
Retorna *True* se o argumento for subnormal, e *False* caso contrário.

is_zero()
Retorna *True* se o argumento for um zero (positivo ou negativo) e *False* caso contrário.

ln(*context=None*)
Retorna o logaritmo (base e) natural do operando. O resultado é arredondado corretamente usando o modo de arredondamento *ROUND_HALF_EVEN*.

log10(*context=None*)
Retorna o logaritmo da base dez do operando. O resultado é arredondado corretamente usando o modo de arredondamento *ROUND_HALF_EVEN*.

logb(*context=None*)
Para um número diferente de zero, retorna o expoente ajustado de seu operando como uma instância de *Decimal*. Se o operando é zero, `Decimal('-Infinity')` é retornado e o sinalizador *DivisionByZero* é levantado. Se o operando for um infinito, `Decimal('Infinity')` será retornado.

logical_and(*other*, *context=None*)
`logic_and()` é uma operação lógica que leva dois *operandos lógicos* (consulte logic_operands_label). O resultado é o `and` dígito a dígito dos dois operandos.

logical_invert(*context=None*)
logical_invert() é uma operação lógica. O resultado é a inversão dígito a dígito do operando.

logical_or(*other*, *context=None*)
`logic_or()` é uma operação lógica que leva dois *operandos lógicos* (consulte logic_operands_label). O resultado é o `or` dígito a dígito dos dois operandos.

logical_xor(*other*, *context=None*)
`logical_xor()` é uma operação lógica que leva dois *operandos lógicos* (consulte logic_operands_label). O resultado é o ou exclusivo dígito a dígito ou dos dois operandos.

max(*other*, *context=None*)
Semelhante a `max(self, other)`, exceto que a regra de arredondamento de contexto é aplicada antes de retornar e que os valores `NaN` são sinalizados ou ignorados (dependendo do contexto e se estão sinalizando ou silenciosos).

max_mag(*other*, *context=None*)
Semelhante ao método *`max()`*, mas a comparação é feita usando os valores absolutos dos operandos.

min(*other*, *context=None*)
Semelhante a `min(self, other)`, exceto que a regra de arredondamento de contexto é aplicada antes de retornar e que os valores `NaN` são sinalizados ou ignorados (dependendo do contexto e se estão sinalizando ou silenciosos).

min_mag(*other*, *context=None*)
Semelhante ao método *`min()`*, mas a comparação é feita usando os valores absolutos dos operandos.

next_minus(*context=None*)
Retorna o maior número representável no contexto fornecido (ou no contexto atual da thread, se nenhum contexto for fornecido) que seja menor que o operando especificado.

next_plus(*context=None*)
Retorna o menor número representável no contexto fornecido (ou no contexto atual da thread, se nenhum contexto for fornecido) que seja maior que o operando fornecido.

next_toward(*other*, *context=None*)
Se os dois operandos forem desiguais, retorna o número mais próximo ao primeiro operando na direção do segundo operando. Se os dois operandos forem numericamente iguais, retorne uma cópia do primeiro operando com o sinal configurado para ser o mesmo que o sinal do segundo operando.

normalize(*context=None*)
Normaliza o número eliminando os zeros à direita e convertendo qualquer resultado igual a `Decimal('0')` para `Decimal('0e0')`. Usado para produzir valores canônicos para atributos de uma classe de equivalência. Por exemplo, `Decimal('32.100')` e `Decimal('0.321000e+2')` normalizam com o valor equivalente `Decimal('32.1')`.

number_class(*context=None*)
Retorna uma string descrevendo a *classe* do operando. O valor retornado é uma das dez sequências a seguir.

- `"-Infinity"`, indicando que o operando é infinito negativo.
- `"-Normal"`, indicando que o operando é um número normal negativo.
- `"-Subnormal"`, indicando que o operando é negativo e subnormal.
- `"-Zero"`, indicando que o operando é um zero negativo.
- `"+Zero"`, indicando que o operando é um zero positivo.
- `"+Subnormal"`, indicando que o operando é positivo e subnormal.
- `"+Normal"`, indicando que o operando é um número normal positivo.
- `"+Infinity"`, indicando que o operando é infinito positivo.
- `"NaN"`, indicando que o operando é um NaN silencioso (não é um número).

- `"sNaN"`, indicando que o operando é um NaN sinalizador.

quantize(*exp*, *rounding=None*, *context=None*)
Retorna um valor igual ao primeiro operando após o arredondamento e com o expoente do segundo operando.

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

Diferentemente de outras operações, se o comprimento do coeficiente após a operação de quantização for maior que a precisão, então *InvalidOperation* é sinalizado. Isso garante que, a menos que haja uma condição de erro, o expoente quantizado é sempre igual ao do operando do lado direito.

Também, diferentemente de outras operações, a quantização nunca sinaliza Underflow, mesmo que o resultado seja subnormal e inexato.

Se o expoente do segundo operando for maior que o do primeiro, o arredondamento poderá ser necessário. Nesse caso, o modo de arredondamento é determinado pelo argumento `rounding`, se fornecido, ou pelo argumento `context` fornecido; se nenhum argumento for fornecido, o modo de arredondamento do contexto da thread atual será usado.

Um erro é retornado sempre que o expoente resultante for maior que `Emax` ou menor que `Etiny`.

radix()
Retorna `Decimal(10)`, a raiz (base) na qual a classe *Decimal* faz toda a sua aritmética. Incluído para compatibilidade com a especificação.

remainder_near(*other*, *context=None*)
Retorna o resto da divisão de *self* por *other*. Isso é diferente de `self % other`, pois o sinal do resto é escolhido para minimizar seu valor absoluto. Mais precisamente, o valor de retorno é `self - n * other`, onde `n` é o número inteiro mais próximo do valor exato de `self / other`, e se dois números inteiros estiverem igualmente próximos, o par será é escolhido.

Se o resultado for zero, seu sinal será o sinal de *self*.

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate(*other*, *context=None*)
Retorna o resultado da rotação dos dígitos do primeiro operando em uma quantidade especificada pelo segundo operando. O segundo operando deve ser um número inteiro no intervalo - precisão através da precisão. O valor absoluto do segundo operando fornece o número de locais a serem rotacionados. Se o segundo operando for positivo, a rotação será para a esquerda; caso contrário, a rotação será para a direita. O coeficiente do primeiro operando é preenchido à esquerda com zeros na precisão do comprimento, se necessário. O sinal e o expoente do primeiro operando não são alterados.

same_quantum(*other*, *context=None*)
Testa se “self” e “other” têm o mesmo expoente ou se ambos são `NaN`.

Esta operação não é afetada pelo contexto e é silenciosa: nenhum sinalizador é alterado e nenhum arredondamento é executado. Como uma exceção, a versão C pode levantar InvalidOperation se o segundo operando não puder ser convertido exatamente.

scaleb(*other*, *context=None*)
Retorna o primeiro operando com o expoente ajustado pelo segundo. Da mesma forma, retorna o primeiro operando multiplicado por `10**other`. O segundo operando deve ser um número inteiro.

shift(*other*, *context=None*)
Retorna o resultado da troca dos dígitos do primeiro operando em uma quantidade especificada pelo segundo operando. O segundo operando deve ser um número inteiro no intervalo - precisão através da precisão. O valor absoluto do segundo operando fornece o número de locais a serem deslocados. Se o segundo operando for positivo, o deslocamento será para a esquerda; caso contrário, a mudança é para a direita. Os dígitos deslocados para o coeficiente são zeros. O sinal e o expoente do primeiro operando não são alterados.

sqrt(*context=None*)
Retorna a raiz quadrada do argumento para a precisão total.

to_eng_string(*context=None*)
Converte em uma string, usando notação de engenharia, se for necessário um expoente.

A notação de engenharia possui um expoente que é múltiplo de 3. Isso pode deixar até 3 dígitos à esquerda da casa decimal e pode exigir a adição de um ou dois zeros à direita.

Por exemplo, isso converte `Decimal('123E+1')` para `Decimal('1.23E+3')`.

to_integral(*rounding=None*, *context=None*)
Idêntico ao método *to_integral_value()*. O nome `to_integral` foi mantido para compatibilidade com versões mais antigas.

to_integral_exact(*rounding=None*, *context=None*)
Arredonda para o número inteiro mais próximo, sinalizando *Inexact* ou *Rounded*, conforme apropriado, se o arredondamento ocorrer. O modo de arredondamento é determinado pelo parâmetro *rouding*‘, se fornecido, ou pelo `context` especificado. Se nenhum parâmetro for fornecido, o modo de arredondamento do contexto atual será usado.

to_integral_value(*rounding=None*, *context=None*)
Arredonda para o número inteiro mais próximo sem sinalizar *Inexact* ou `Rounding`. Se fornecido, aplica *rounding*; caso contrário, usa o método de arredondamento no *context* especificado ou no contexto atual.

Operandos lógicos

Os métodos `logical_and()`, `logical_invert()`, `logical_or()` e `logical_xor()` esperam que seus argumentos sejam *operandos lógicos*. Um *operando lógico* é uma instância de *Decimal* cujo expoente e sinal são zero e cujos dígitos são todos `0` ou `1`.

9.4.3 Objetos de contexto

Contextos são ambientes para operações aritméticas. Eles governam a precisão, estabelecem regras para arredondamento, determinam quais sinais são tratados como exceções e limitam o alcance dos expoentes.

Cada thread possui seu próprio contexto atual que é acessado ou alterado usando as funções *getcontext()* e *setcontext()*:

`decimal`.**getcontext**()
Retorna o contexto atual para a thread ativa.

`decimal`.**setcontext**(*c*)
Define o contexto atual para a thread ativa como *C*.

Você também pode usar a instrução `with` e a função func:*localcontext* para alterar temporariamente o contexto ativo.

`decimal`.**localcontext**(*ctx=None*)
Retorna um gerenciador de contexto que vai definir o contexto atual da thread ativa para uma cópia de *ctx* na entrada da instrução “with” e restaurar o contexto anterior ao sair da instrução “with”. Se nenhum contexto for especificado, uma cópia do contexto atual será usada.

Por exemplo, o código a seguir define a precisão decimal atual para 42 casas, executa um cálculo e restaura automaticamente o contexto anterior:

```
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42   # Perform a high precision calculation
    s = calculate_something()
s = +s  # Round the final result back to the default precision
```

Novos contextos também podem ser criados usando o construtor *Context* descrito abaixo. Além disso, o módulo fornece três contextos pré-criados:

class decimal.**BasicContext**
Este é um contexto padrão definido pela Especificação Aritmética Decimal Geral. A precisão está definida como nove. O arredondamento está definido como *ROUND_HALF_UP*. Todos os sinalizadores são limpos. Todos as armadilhas estão ativadas (tratadas como exceções), exceto por *Inexact*, *Rounded* e *Subnormal*.

Como muitas das armadilhas estão ativadas, esse contexto é útil para depuração.

class decimal.**ExtendedContext**
Este é um contexto padrão definido pela Especificação Aritmética Decimal Geral. A precisão está definida como nove. O arredondamento está definido como *ROUND_HALF_EVEN*. Todos os sinalizadores estão limpos. Nenhuma armadilha está ativada (de forma que exceções não são levantadas durante os cálculos).

Como as armadilhas estão desativadas, esse contexto é útil para aplicativos que preferem ter o valor de resultado de `NaN` ou `Infinity` em vez de levantar exceções. Isso permite que um aplicativo conclua uma execução na presença de condições que interromperiam o programa.

class decimal.**DefaultContext**
Este contexto é usado pelo construtor *Context* como um protótipo para novos contextos. Alterar um campo (tal como precisão) tem o efeito de alterar o padrão para novos contextos criados pelo construtor *Context*.

Esse contexto é mais útil em ambientes multithread. A alteração de um dos campos antes do início das threads tem o efeito de definir os padrões para todo o sistema. Não é recomendável alterar os campos após o início das threads, pois exigiria sincronização de threads para evitar condições de corrida.

Em ambientes de thread única, é preferível não usar esse contexto. Em vez disso, basta criar contextos explicitamente, conforme descrito abaixo.

Os valores padrão são `prec=28`, `rounding=`*ROUND_HALF_EVEN* e armadilhas ativadas para *Overflow*, *InvalidOperation* e *DivisionByZero*.

Além dos três contextos fornecidos, novos contextos podem ser criados com o construtor *Context*.

class decimal.**Context**(*prec=None*, *rounding=None*, *Emin=None*, *Emax=None*, *capitals=None*, *clamp=None*, *flags=None*, *traps=None*)
Cria um novo contexto. Se um campo não for especificado ou for *None*, os valores padrão serão copiados de *DefaultContext*. Se o campo *flags* não for especificado ou for *None*, todos os sinalizadores serão limpados.

prec é um número inteiro no intervalo [`1`, *MAX_PREC*] que define a precisão das operações aritméticas no contexto.

A opção *rounding* é uma das constantes listadas na seção *Modos de arredondamento*.

Os campos *traps* e *flags* listam todos os sinais a serem configurados. Geralmente, novos contextos devem apenas definir armadilhas e deixar os sinalizadores limpos.

Os campos *Emin* e *Emax* são números inteiros que especificam os limites externos permitidos para expoentes. *Emin* deve estar no intervalo [*MIN_EMIN*, `0`], *Emax* no intervalo [`0`, *MAX_EMAX*].

O campo *capitals* é `0` ou `1` (o padrão). Se definido como `1`, os expoentes serão impressos com um `E` maiúsculo; caso contrário, um `e` minúscula é usado: `Decimal('6.02e+23')`.

O campo *clamp* é `0` (o padrão) ou `1`. Se definido como `1`, o expoente `e` de uma instância de *Decimal* representável nesse contexto é estritamente limitado ao intervalo `Emin - prec + 1 <= e <= Emax - prec + 1`. Se *clamp* for `0`, uma condição mais fraca será mantida: o expoente ajustado da instância de *Decimal* é no máximo `Emax`. Quando *clamp* é `1`, um grande número normal terá, sempre que possível, seu expoente reduzido e um número correspondente de zeros adicionado ao seu coeficiente, para ajustar as restrições do expoente; isso preserva o valor do número, mas perde informações sobre zeros à direita significativos. Por exemplo:

```
>>> Context(prec=6, Emax=999, clamp=1).create_decimal('1.23e999')
Decimal('1.23000E+999')
```

Um valor de *clamp* de `1` permite compatibilidade com os formatos de intercâmbio decimal de largura fixa especificados na IEEE 754.

A classe *Context* define vários métodos de uso geral, bem como um grande número de métodos para fazer aritmética diretamente em um determinado contexto. Além disso, para cada um dos métodos de *Decimal* descritos acima (com exceção dos métodos `adjusted()` e `as_tuple()`) existe um método correspondente em *Context*. Por exemplo, para uma instância `C` de *Context* e uma instância `x` de *Decimal*, `C.exp(x)` é equivalente a `x.exp(context=C)`. Cada método de *Context* aceita um número inteiro do Python (uma instância de *int*) em qualquer lugar em que uma instância de Decimal seja aceita.

clear_flags()
Redefine todos os sinalizadores para `0`.

clear_traps()
Redefine todas as armadilhas para `0`.

Novo na versão 3.3.

copy()
Retorna uma duplicata do contexto.

copy_decimal(*num*)
Retorna uma cópia de uma instância de Decimal *num*.

create_decimal(*num*)
Cria uma nova instância decimal a partir de *num*, mas usando *self* como contexto. Diferentemente do construtor de *Decimal*, a precisão do contexto, o método de arredondamento, os sinalizadores e as armadilhas são aplicadas à conversão.

Isso é útil porque as constantes geralmente são fornecidas com uma precisão maior do que a necessária pelo aplicativo. Outro benefício é que o arredondamento elimina imediatamente os efeitos indesejados dos dígitos além da precisão atual. No exemplo a seguir, o uso de entradas não arredondadas significa que adicionar zero a uma soma pode alterar o resultado:

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

Este método implementa a operação "to-number" da especificação IBM. Se o argumento for uma sequência, nenhum espaço em branco à esquerda ou à direita ou sublinhado serão permitidos.

create_decimal_from_float(*f*)
Cria uma nova instância de Decimal a partir de um ponto flutuante *f*, mas arredondando usando *self* como contexto. Diferentemente do método da classe *Decimal.from_float()*, a precisão do contexto, o método de arredondamento, os sinalizadores e as armadilhas são aplicados à conversão.

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
    ...
decimal.Inexact: None
```

Novo na versão 3.1.

Etiny()
Retorna um valor igual a `Emin - prec + 1`, que é o valor mínimo do expoente para resultados subnormais. Quando ocorre o estouro negativo, o expoente é definido como *Etiny*.

Etop()
Retorna um valor igual a `Emax - prec + 1`.

A abordagem usual para trabalhar com decimais é criar instâncias de *Decimal* e depois aplicar operações aritméticas que ocorrem no contexto atual da thread ativa. Uma abordagem alternativa é usar métodos de contexto para calcular dentro de um contexto específico. Os métodos são semelhantes aos da classe *Decimal* e são contados apenas brevemente aqui.

abs(*x*)
Retorna o valor absoluto de *x*.

add(*x*, *y*)
Retorna a soma de *x* e *y*.

canonical(*x*)
Retorna o mesmo objeto de Decimal *x*.

compare(*x*, *y*)
Compara *x* e *y* numericamente.

compare_signal(*x*, *y*)
Compara os valores dos dois operandos numericamente.

compare_total(*x*, *y*)
Compara dois operandos usando sua representação abastrata.

compare_total_mag(*x*, *y*)
Compara dois operandos usando sua representação abstrata, ignorando o sinal.

copy_abs(*x*)
Retorna uma cópia de *x* com o sinal definido para 0.

copy_negate(*x*)
Retorna uma cópia de *x* com o sinal invertido.

copy_sign(*x*, *y*)
Copia o sinal de *y* para *x*.

divide(*x*, *y*)
Retorna *x* dividido por *y*.

divide_int(*x*, *y*)
Retorna *x* dividido por *y*, truncado para um inteiro.

divmod(*x*, *y*)
Divide dois números e retorna a parte inteira do resultado.

exp(*x*)
Retorna *e* ** *x*.

fma(*x*, *y*, *z*)
Retorna *x* multiplicado por *y*, mais *z*.

is_canonical(*x*)
Retorna `True` se *x* for canonical; do contrário, retorna `False`.

is_finite(*x*)
Retorna `True` se *x* for finito; do contrário, retorna `False`.

is_infinite(*x*)
Retorna `True` se *x* for infinito; caso contrário, retorna `False`.

is_nan(*x*)
Retorna `True` se *x* for qNaN ou sNaN; caso contrário, retorna `False`.

is_normal(*x*)
Retorna `True` se *x* for um número normal; do contrário, retorna `False`.

is_qnan(*x*)
Retorna `True` se *x* for um NaN silencioso; caso contrário, retorna `False`.

is_signed(*x*)
Retorna `True` se *x* for negativo; do contrário, retorna `False`.

is_snan(*x*)
Retorna `True` se *x* for um NaN sinalizador; caso contrário, retorna `False`.

is_subnormal(*x*)
Retorna `True` se *x* for subnormal; caso contrário, retorna `False`.

is_zero(*x*)
Retorna `True` se *x* for zero; caso contrário, retorna `False`.

ln(*x*)
Retorna o logaritmo natural (base e) de *x*.

log10(*x*)
Retorna o logaritmo de base 10 de *x*.

logb(*x*)
Retorna o expoente da magnitude do MSD do operando.

logical_and(*x*, *y*)
Aplica a operação lógica *e* entre cada dígito do operando.

logical_invert(*x*)
Inverte todos os dígitos em *x*.

logical_or(*x*, *y*)
Aplica a operação lógica *ou* entre cada dígito do operando.

logical_xor(*x*, *y*)
Aplica a operação lógica *ou eclusivo* entre cada dígito do operando.

max(*x*, *y*)
Compara dois valores numericamente e retorna o máximo.

max_mag(*x*, *y*)
Compara dois valores numericamente com seu sinal ignorado.

min(*x*, *y*)
Compara dois valores numericamente e retorna o mínimo.

min_mag(*x*, *y*)
Compara dois valores numericamente com seu sinal ignorado.

minus(*x*)
Minus corresponde ao operador de subtração de prefixo unário no Python.

multiply(*x*, *y*)
Retorna o produto de *x* e *y*.

next_minus(*x*)
Retorna o maior número representável menor que *x*.

next_plus(*x*)
Retorna o menor número representável maior que *x*.

next_toward(*x*, *y*)
Retorna o número mais próximo a *x*, em direção a *y*.

normalize(*x*)
Reduz *x* para sua forma mais simples.

number_class(*x*)
Retorna uma indicação da classe de *x*.

plus(*x*)
Plus corresponde ao operador de soma de prefixo unário no Python. Esta operação aplica a precisão e o arredondamento do contexto, portanto *não* é uma operação de identidade.

power(*x*, *y*, *modulo=None*)
Retorna `x` à potência de `y`, com a redução de módulo `modulo` if given.

Com dois argumentos, calcula `x**y`. Se `x` for negativo, `y` deve ser inteiro. O resultado será inexato, a menos que `y` seja inteiro e o resultado seja finito e possa ser expresso exatamente em dígitos de "precisão". O modo de arredondamento do contexto é usado. Os resultados são sempre arredondados corretamente na versão Python.

Alterado na versão 3.3: O módulo C calcula *power()* em termos das funções corretamente arredondadas *exp()* e *ln()*. O resultado é bem definido, mas apenas "quase sempre corretamente arredondado".

Com três argumentos, calcula `(x**y) % modulo`. Para o formulário de três argumentos, as seguintes restrições nos argumentos são válidas:

- todos os três argumentos devem ser inteiros
- `y` não pode ser negativo
- pelo menos um de `x` ou `y` não pode ser negativo
- `modulo` não pode ser zero e deve ter mais dígitos de "precisão"

O valor resultante de `Context.power(x, y, modulo)` é igual ao valor que seria obtido ao computar `(x**y) % modulo` com precisão ilimitada, mas é calculado com mais eficiência . O expoente do resultado é zero, independentemente dos expoentes de `x`, `y` e `modulo`. O resultado é sempre exato.

quantize(*x*, *y*)
Retorna um valor igual a *x* (arredondado), tendo o expoente de *y*.

radix()
Só retorna 10, já que isso é Decimal, :)

remainder(*x*, *y*)
Retorna o resto da divisão inteira.

O sinal do resultado, se diferente de zero, é o mesmo que o do dividendo original.

remainder_near(*x*, *y*)
Retorna `x - y * n`, onde *n* é o número inteiro mais próximo do valor exato de `x / y` (se o resultado for 0, seu sinal será o sinal de *x*).

rotate(*x*, *y*)
Retorna uma cópia girada de *x*, *y* vezes.

same_quantum(*x*, *y*)
Retorna `True` se os dois operandos tiverem o mesmo expoente.

scaleb(*x*, *y*)
Retorna o primeiro operando após adicionar o segundo valor sua exp.

shift(*x*, *y*)
Retorna uma cópia deslocada de *x*, *y* vezes.

sqrt(*x*)
Raiz quadrada de um número não negativo para precisão do contexto.

subtract(*x*, *y*)
Retorna a diferença entre *x* e *y*.

to_eng_string(*x*)
Converte em uma string, usando notação de engenharia, se for necessário um expoente.

A notação de engenharia possui um expoente que é múltiplo de 3. Isso pode deixar até 3 dígitos à esquerda da casa decimal e pode exigir a adição de um ou dois zeros à direita.

to_integral_exact(*x*)
Arredonda para um número inteiro.

to_sci_string(*x*)
Converte um número em uma string usando notação científica.

9.4.4 Constantes

As constantes nesta seção são relevantes apenas para o módulo C. Eles também estão incluídos na versão pura do Python para compatibilidade.

	32 bits	64 bits
`decimal.`**MAX_PREC**	`425000000`	`999999999999999999`
`decimal.`**MAX_EMAX**	`425000000`	`999999999999999999`
`decimal.`**MIN_EMIN**	`-425000000`	`-999999999999999999`
`decimal.`**MIN_ETINY**	`-849999999`	`-1999999999999999997`

`decimal.`**HAVE_THREADS**
O valor é `True`. Descontinuado porque o Python agora sempre tem threads.

Obsoleto desde a versão 3.9.

decimal.**HAVE_CONTEXTVAR**
: O valor padrão é `True`. Se o Python for compilado usando `--without-decimal-contextvar`, a versão C usará um contexto local de thread em vez de local de corrotina e o valor será `False`. Isso é um pouco mais rápido em alguns cenários de contexto aninhados.

Novo na versão 3.9: backport realizado para 3.7 e 3.8

9.4.5 Modos de arredondamento

decimal.**ROUND_CEILING**
: Arredonda para `Infinity`.

decimal.**ROUND_DOWN**
: Arredonda para zero.

decimal.**ROUND_FLOOR**
: Arredonda para `-Infinity`.

decimal.**ROUND_HALF_DOWN**
: Arrendonda para o mais próximo com empate tendendo a zero.

decimal.**ROUND_HALF_EVEN**
: Arredonda para o mais próximo com empates indo para o mais próximo inteiro par.

decimal.**ROUND_HALF_UP**
: Arrendonda para o mais próximo com empates se afastando de zero.

decimal.**ROUND_UP**
: Arredonda se afastando de zero.

decimal.**ROUND_05UP**
: Arredonda se afastando de zero se o último dígito após o arredondamento para zero fosse 0 ou 5; caso contrário, arredonda para zero.

9.4.6 Sinais

Sinais representam condições que surgem durante o cálculo. Cada um corresponde a um sinalizador de contexto e um ativador de armalhida de contexto.

O sinalizador de contexto é definido sempre que a condição é encontrada. Após o cálculo, os sinalizadores podem ser verificados para fins informativos (por exemplo, para determinar se um cálculo era exato). Depois de verificar os sinalizadores, certifique-se de limpar todos os sinalizadores antes de iniciar o próximo cálculo.

Se o ativador de armadilha de contexto estiver definido para o sinal, a condição fará com que uma exceção Python seja levantada. Por exemplo, se a armadilha *DivisionByZero* for configurada, uma exceção *DivisionByZero* será levantada ao encontrar a condição.

class decimal.**Clamped**
: Alterou um expoente para ajustar as restrições de representação.

  Normalmente, a fixação ocorre quando um expoente fica fora dos limites do contexto `Emin` e attr:*Emax*. Se possível, o expoente é reduzido para caber adicionando zeros ao coeficiente.

class decimal.**DecimalException**
: Classe base para outros sinais e uma subclasse de *ArithmeticError*.

class decimal.**DivisionByZero**
Sinaliza a divisão de um número não infinito por zero.

Pode ocorrer com divisão, divisão de módulo ou ao elevar um número a uma potência negativa. Se este sinal não for capturado, retornará `Infinity` ou `-Infinity` com o sinal determinado pelas entradas do cálculo.

class decimal.**Inexact**
Indica que o arredondamento ocorreu e o resultado não é exato.

Sinaliza quando dígitos diferentes de zero foram descartados durante o arredondamento. O resultado arredondado é retornado. O sinalizador ou armadilha de sinal é usado para detectar quando os resultados são inexatos.

class decimal.**InvalidOperation**
Uma operação inválida foi realizada.

Indica que uma operação foi solicitada que não faz sentido. Se não for capturado, retorna `NaN`. As possíveis causas incluem:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

class decimal.**Overflow**
Estouro numérico

Indica que o expoente é maior que `Emax` após o arredondamento ocorrer. Se não for capturado, o resultado depende do modo de arredondamento, puxando para dentro para o maior número finito representável ou arredondando para fora para `Infinity`. Nos dois casos, *`Inexact`* e *`Rounded`* também são sinalizados.

class decimal.**Rounded**
O arredondamento ocorreu, embora possivelmente nenhuma informação tenha sido perdida.

Sinalizado sempre que o arredondamento descarta dígitos; mesmo que esses dígitos sejam zero (como arredondamento `5.00` a `5.0`). Se não for capturado, retorna o resultado inalterado. Este sinal é usado para detectar a perda de dígitos significativos.

class decimal.**Subnormal**
O expoente foi menor que `Emin` antes do arredondamento.

Ocorre quando um resultado da operação é subnormal (o expoente é muito pequeno). Se não for capturado, retorna o resultado inalterado.

class decimal.**Underflow**
Estouro negativo numérico com resultado arredondado para zero.

Ocorre quando um resultado subnormal é empurrado para zero arredondando. *`Inexact`* e *`Subnormal`* também são sinalizados.

class decimal.**FloatOperation**
Ativa semânticas mais rigorosas para misturar carros alegóricos e decimais.

Se o sinal não for capturado (padrão), a mistura de tipos float e Decimal será permitida no construtor class:*~decimal.Decimal* constructor, *`create_decimal()`* e em todos os operadores de comparação. Tanto a conversão quanto as comparações são exatas. Qualquer ocorrência de uma operação mista é registrada silenciosamente pela configuração *`FloatOperation`* nos sinalizadores de contexto. Conversões explícitas com *`from_float()`* or *`create_decimal_from_float()`* não definem o sinalizador.

Caso contrário (o sinal é capturado), apenas comparações de igualdade e conversões explícitas são silenciosas. Todas as outras operações mistas aumentam *FloatOperation*.

A tabela a seguir resume a hierarquia de sinais:

```
exceptions.ArithmeticError(exceptions.Exception)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
        FloatOperation(DecimalException, exceptions.TypeError)
```

9.4.7 Observações sobre ponto flutuante

Atenuando o erro de arredondamento com maior precisão

O uso do ponto flutuante decimal elimina o erro de representação decimal (possibilitando representar `0.1` de forma exata); no entanto, algumas operações ainda podem sofrer erros de arredondamento quando dígitos diferentes de zero excederem a precisão fixa.

Os efeitos do erro de arredondamento podem ser amplificados pela adição ou subtração de quantidades quase compensadoras, resultando em perda de significância. Knuth fornece dois exemplos instrutivos em que a aritmética de ponto flutuante arredondado com precisão insuficiente causa a quebra das propriedades associativas e distributivas da adição:

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

O módulo *decimal* permite restaurar as identidades expandindo a precisão o suficiente para evitar perda de significância:

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
```

(continua na próxima página)

(continuação da página anterior)

```
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')
```

Valores especiais

O sistema numérico para o módulo *decimal* fornece valores especiais, incluindo `NaN`, `sNaN`, `-Infinity`, `Infinity` e dois zeros, `+0` e `-0`.

Os infinitos podem ser construídos diretamente com: `Decimal('Infinity')`. Além disso, eles podem resultar da divisão por zero quando o sinal *DivisionByZero* não é capturado. Da mesma forma, quando o sinal *Overflow* não é capturado, o infinito pode resultar do arredondamento além dos limites do maior número representável.

Os infinitos contêm sinais (afins) e podem ser usados em operações aritméticas, onde são tratados como números muito grandes e indeterminados. Por exemplo, adicionar uma constante ao infinito fornece outro resultado infinito.

Algumas operações são indeterminadas e retornam `NaN` ou, se o sinal *InvalidOperation* for capturado, levanta uma exceção. Por exemplo, `0/0` retorna `NaN`, que significa "não é um número" em inglês. Esta variação de `NaN` é silenciosa e, uma vez criada, fluirá através de outros cálculos sempre resultando em outra `NaN`. Esse comportamento pode ser útil para uma série de cálculos que ocasionalmente têm entradas ausentes — ele permite que o cálculo continue enquanto sinaliza resultados específicos como inválidos.

Uma variante é `sNaN`, que sinaliza em vez de permanecer em silêncio após cada operação. Esse é um valor de retorno útil quando um resultado inválido precisa interromper um cálculo para tratamento especial.

O comportamento dos operadores de comparação do Python pode ser um pouco surpreendente onde um `NaN` está envolvido. Um teste de igualdade em que um dos operandos é um `NaN` silencioso ou sinalizador sempre retorna *False* (mesmo ao fazer `Decimal('NaN')==Decimal('NaN')`), enquanto um teste de desigualdade sempre retorna *True*. Uma tentativa de comparar dois decimais usando qualquer um dos operadores <, <=, > ou >= levantará o sinal *InvalidOperation* se um dos operandos for um `NaN` e retorna *False* se esse sinal não for capturado. Observe que a especificação aritmética decimal geral não especifica o comportamento das comparações diretas; estas regras para comparações envolvendo a `NaN` foram retiradas do padrão IEEE 854 (consulte a Tabela 3 na seção 5.7). Para garantir uma rígida conformidade com os padrões, use os métodos `compare()` e `compare-signal()`.

Os zeros com sinais podem resultar de cálculos insuficientes. Eles mantêm o sinal que teria resultado se o cálculo tivesse sido realizado com maior precisão. Como sua magnitude é zero, os zeros positivos e negativos são tratados como iguais e seu sinal é informativo.

Além dos dois zeros com sinais que são distintos e iguais, existem várias representações de zero com diferentes precisões e ainda com valor equivalente. Isso leva um pouco de tempo para se acostumar. Para um olho acostumado a representações de ponto flutuante normalizadas, não é imediatamente óbvio que o seguinte cálculo retorne um valor igual a zero:

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```

9.4.8 Trabalhando com threads

A função *getcontext()* acessa um objeto *Context* diferente para cada thread. Ter contextos de threads separadas significa que as threads podem fazer alterações (como `getcontext().prec=10`) sem interferir em outros threads.

Da mesma forma, a função *setcontext()* atribui automaticamente seu destino à thread atual.

Se *setcontext()* não tiver sido chamado antes de *getcontext()*, então *getcontext()* criará automaticamente um novo contexto para uso na thread atual.

O novo contexto é copiado de um contexto protótipo chamado *DefaultContext*. Para controlar os padrões para que cada thread, use os mesmos valores em todo o aplicativo, modifique diretamente o objeto *DefaultContext*. Isso deve ser feito *antes* de qualquer thread ser iniciada, para que não haja uma condição de corrida entre as threads chamando *getcontext()*. Por exemplo:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
 . . .
```

9.4.9 Receitas

Aqui estão algumas receitas que servem como funções utilitárias e que demonstram maneiras de trabalhar com a classe *Decimal*:

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """Convert Decimal to a money formatted string.

    places:  required number of places after the decimal point
    curr:    optional currency symbol before the sign (may be blank)
    sep:     optional grouping separator (comma, period, space, or blank)
    dp:      decimal point indicator (comma or period)
             only specify as blank when places is zero
    pos:     optional sign for positive numbers: '+', space or blank
    neg:     optional sign for negative numbers: '-', '(', space or blank
    trailneg:optional trailing minus indicator:  '-', ')', space or blank

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
```

(continua na próxima página)

(continuação da página anterior)

```
    '<0.02>'

    """
    q = Decimal(10) ** -places      # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = list(map(str, digits))
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        build(next() if digits else '0')
    if places:
        build(dp)
    if not digits:
        build('0')
    i = 0
    while digits:
        build(next())
        i += 1
        if i == 3 and digits:
            i = 0
            build(sep)
    build(curr)
    build(neg if sign else pos)
    return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision.

    >>> print(pi())
    3.141592653589793238462643383

    """
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)      # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s               # unary plus applies the new precision

def exp(x):
    """Return e raised to the power of x.  Result type matches input type.

    >>> print(exp(Decimal(1)))
    2.718281828459045235360287471
    >>> print(exp(Decimal(2)))
    7.389056098930650227230427461
    >>> print(exp(2.0))
    7.38905609893
    >>> print(exp(2+0j))
    (7.38905609893+0j)
```

(continua na próxima página)

(continuação da página anterior)

```
    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
        num *= x
        s += num / fact
    getcontext().prec -= 2
    return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(cos(Decimal('0.5')))
    0.8775825618903727161162815826
    >>> print(cos(0.5))
    0.87758256189
    >>> print(cos(0.5+0j))
    (0.87758256189+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """Return the sine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(sin(Decimal('0.5')))
    0.4794255386042030002732879352
    >>> print(sin(0.5))
    0.479425538604
    >>> print(sin(0.5+0j))
    (0.479425538604+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
    while s != lasts:
        lasts = s
```

(continua na próxima página)

(continuação da página anterior)

```
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s
```

9.4.10 Perguntas Frequentes sobre o Decimal

P. É complicado digitar `decimal.Decimal('1234.5')`. Existe uma maneira de minimizar a digitação ao usar o interpretador interativo?

R. Alguns usuários abreviam o construtor para apenas uma única letra:

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

P. Em um aplicativo de ponto fixo com duas casas decimais, algumas entradas têm muitas casas e precisam ser arredondadas. Outros não devem ter dígitos em excesso e precisam ser validados. Quais métodos devem ser usados?

Q. O método `quantize()` arredonda para um número fixo de casas decimais. Se a armadilha *Inexact* estiver configurada, também será útil para validação:

```
>>> TWOPLACES = Decimal(10) ** -2       # same as Decimal('0.01')
```

```
>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
```

```
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')
```

```
>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
   ...
Inexact: None
```

P. Depois de validar entradas de duas casas, como mantenho essa invariável em uma aplicação?

R. Algumas operações como adição, subtração e multiplicação por um número inteiro preservam automaticamente o ponto fixo. Outras operações, como divisão e multiplicação não inteira, alteram o número de casas decimais e precisam ser seguidas com uma etapa `quantize()`:

```
>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                           # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                          # So does integer multiplication
Decimal('4314.24')
```

(continua na próxima página)

(continuação da página anterior)

```
>>> (a * b).quantize(TWOPLACES)      # Must quantize non-integer multiplication
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES)      # And quantize division
Decimal('0.03')
```

No desenvolvimento de aplicativos de ponto fixo, é conveniente definir funções para lidar com a etapa `quantize()`:

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)
```

```
>>> mul(a, b)                        # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

P. Existem várias maneiras de expressar o mesmo valor. Os números `200`, `200.000`, const:*2E2* e `02E+4` têm todos o mesmo valor em várias precisões. Existe uma maneira de transformá-los em um único valor canônico reconhecível?

R. O método `normalize()` mapeia todos os valores equivalentes para um único representativo:

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

P. Alguns valores decimais sempre são impressos com notação exponencial. Existe uma maneira de obter uma representação não exponencial?

R. Para alguns valores, a notação exponencial é a única maneira de expressar o número de casas significativas no coeficiente. Por exemplo, expressar `5.0E+3` como `5000` mantém o valor constante, mas não pode mostrar a significância de duas casa do original.

Se uma aplicação não se importa com o rastreamento da significância, é fácil remover o expoente e os zeros à direita, perdendo a significância, mas mantendo o valor inalterado:

```
>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

```
>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```

P. Existe uma maneira de converter um float comum em um *Decimal*?

R. Sim, qualquer número de ponto flutuante binário pode ser expresso exatamente como um Decimal, embora uma conversão exata possa exigir mais precisão do que a intuição sugere:

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

P. Em um cálculo complexo, como posso ter certeza de que não obtive um resultado falso devido à precisão insuficiente ou a anomalias de arredondamento.

R. O módulo decimal facilita o teste de resultados. Uma prática recomendada é executar novamente os cálculos usando maior precisão e com vários modos de arredondamento. Resultados amplamente diferentes indicam precisão insuficiente, problemas no modo de arredondamento, entradas mal condicionadas ou um algoritmo numericamente instável.

P. Notei que a precisão do contexto é aplicada aos resultados das operações, mas não às entradas. Há algo a observar ao misturar valores de diferentes precisões?

R. Sim. O princípio é que todos os valores são considerados exatos, assim como a aritmética desses valores. Somente os resultados são arredondados. A vantagem das entradas é que "o que você vê é o que você obtém". Uma desvantagem é que os resultados podem parecer estranhos se você esquecer que as entradas não foram arredondadas:

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

A solução é aumentar a precisão ou forçar o arredondamento das entradas usando a operação unária de mais:

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')      # unary plus triggers rounding
Decimal('1.23')
```

Como alternativa, as entradas podem ser arredondadas na criação usando o método *Context.create_decimal()*:

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

P. A implementação do CPython é rápida para números grandes?

A. Yes. In the CPython and PyPy3 implementations, the C/CFFI versions of the decimal module integrate the high speed libmpdec library for arbitrary precision correctly-rounded decimal floating point arithmetic. `libmpdec` uses Karatsuba multiplication for medium-sized numbers and the Number Theoretic Transform for very large numbers. However, to realize this performance gain, the context needs to be set for unrounded calculations.

```
>>> c = getcontext()
>>> c.prec = MAX_PREC
>>> c.Emax = MAX_EMAX
>>> c.Emin = MIN_EMIN
```

Novo na versão 3.3.

9.5 `fractions` — Rational numbers

Código Fonte: Lib/fractions.py

The *fractions* module provides support for rational number arithmetic.

A Fraction instance can be constructed from a pair of integers, from another rational number, or from a string.

class fractions.**Fraction**(*numerator=0*, *denominator=1*)
class fractions.**Fraction**(*other_fraction*)
class fractions.**Fraction**(*float*)
class fractions.**Fraction**(*decimal*)
class fractions.**Fraction**(*string*)

The first version requires that *numerator* and *denominator* are instances of *numbers.Rational* and returns a new *Fraction* instance with value `numerator/denominator`. If *denominator* is `0`, it raises a *ZeroDivisionError*. The second version requires that *other_fraction* is an instance of *numbers.Rational* and returns a *Fraction* instance with the same value. The next two versions accept either a *float*

or a *decimal.Decimal* instance, and return a *Fraction* instance with exactly the same value. Note that due to the usual issues with binary floating-point (see tut-fp-issues), the argument to `Fraction(1.1)` is not exactly equal to 11/10, and so `Fraction(1.1)` does *not* return `Fraction(11, 10)` as one might expect. (But see the documentation for the *limit_denominator()* method below.) The last version of the constructor expects a string or unicode instance. The usual form for this instance is:

```
[sign] numerator ['/' denominator]
```

where the optional `sign` may be either '+' or '-' and `numerator` and `denominator` (if present) are strings of decimal digits. In addition, any string that represents a finite value and is accepted by the *float* constructor is also accepted by the *Fraction* constructor. In either form the input string may also have leading and/or trailing whitespace. Here are some examples:

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

The *Fraction* class inherits from the abstract base class *numbers.Rational*, and implements all of the methods and operations from that class. *Fraction* instances are hashable, and should be treated as immutable. In addition, *Fraction* has the following properties and methods:

Alterado na versão 3.2: The *Fraction* constructor now accepts *float* and *decimal.Decimal* instances.

numerator
: Numerator of the Fraction in lowest term.

denominator
: Denominator of the Fraction in lowest term.

from_float(*flt*)
: This class method constructs a *Fraction* representing the exact value of *flt*, which must be a *float*. Beware that `Fraction.from_float(0.3)` is not the same value as `Fraction(3, 10)`.

> **Nota:** From Python 3.2 onwards, you can also construct a *Fraction* instance directly from a *float*.

from_decimal(*dec*)

This class method constructs a *Fraction* representing the exact value of *dec*, which must be a *decimal.Decimal* instance.

Nota: From Python 3.2 onwards, you can also construct a *Fraction* instance directly from a *decimal.Decimal* instance.

---

limit_denominator(*max_denominator=1000000*)

Finds and returns the closest *Fraction* to `self` that has denominator at most max_denominator. This method is useful for finding rational approximations to a given floating-point number:

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

or for recovering a rational number that's represented as a float:

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

__floor__()

Returns the greatest *int* `<= self`. This method can also be accessed through the *math.floor()* function:

```
>>> from math import floor
>>> floor(Fraction(355, 113))
3
```

__ceil__()

Returns the least *int* `>= self`. This method can also be accessed through the *math.ceil()* function.

__round__()

__round__(*ndigits*)

The first version returns the nearest *int* to `self`, rounding half to even. The second version rounds `self` to the nearest multiple of `Fraction(1, 10**ndigits)` (logically, if `ndigits` is negative), again rounding half toward even. This method can also be accessed through the *round()* function.

`fractions.`**gcd**(*a*, *b*)

Return the greatest common divisor of the integers *a* and *b*. If either *a* or *b* is nonzero, then the absolute value of `gcd(a, b)` is the largest integer that divides both *a* and *b*. `gcd(a,b)` has the same sign as *b* if *b* is nonzero; otherwise it takes the sign of *a*. `gcd(0, 0)` returns `0`.

Obsoleto desde a versão 3.5: Use *math.gcd()* instead.

Ver também:

`Módulo :mod:`numbers`` The abstract base classes making up the numeric tower.

9.6 `random` — Gera números pseudoaleatórios

Código Fonte: Lib/random.py

Este módulo implementa geradores de números pseudoaleatórios para várias distribuições.

Para números inteiros, há uma seleção uniforme de um intervalo. Para sequências, há uma seleção uniforme de um elemento aleatório, uma função para gerar uma permutação aleatória de uma lista no local e uma função para amostragem aleatória sem substituição.

Na linha real, existem funções para calcular distribuições uniforme, normal (gaussiana), lognormal, exponencial negativa, gama e beta. Para gerar distribuições de ângulos, a distribuição de von Mises está disponível.

Quase todas as funções do módulo dependem da função básica *`random()`*, que gera um ponto flutuante aleatório uniformemente no intervalo semiaberto [0.0, 1.0). O Python usa o Mersenne Twister como gerador de núcleo. Produz pontos flutuantes de precisão de 53 bits e possui um período de 2**19937-1. A implementação subjacente em C é rápida e segura para threads. O Mersenne Twister é um dos geradores de números aleatórios mais amplamente testados existentes. No entanto, sendo completamente determinístico, não é adequado para todos os fins e é totalmente inadequado para fins criptográficos.

As funções fornecidas por este módulo são, na verdade, métodos vinculados de uma instância oculta da classe *`random.Random`*. Você pode instanciar suas próprias instâncias de *`Random`* para obter geradores que não compartilham estado.

A classe *`Random`* também pode ser usado como subclasse se você quiser usar um gerador básico diferente de sua preferência: nesse caso, substitua os métodos `random()`, `seed()`, `getstate()` e `setstate()`. Opcionalmente, um novo gerador pode fornecer um método `getrandbits()` — isso permite que *`randrange()`* produza seleções a um intervalo grande arbitrário.

O módulo *`random`* também fornece a classe *`SystemRandom`* que usa a função do sistema *`os.urandom()`* para gerar números aleatórios a partir de fontes fornecidas pelo sistema operacional.

> **Aviso:** Os geradores pseudoaleatórios deste módulo não devem ser usados para fins de segurança. Para segurança ou uso criptográfico, veja o módulo *`secrets`*.

Ver também:

M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3–30 1998.

Receita de Complementary-Multiply-with-Carry para um gerador de números aleatórios alternativo compatível com um longo período e operações de atualização comparativamente simples.

9.6.1 Funções de contabilidade

`random.`**`seed`**(*a=None*, *version=2*)

Inicializa o gerador de números aleatórios.

Se *a* for omitido ou `None`, a hora atual do sistema será usada. Se fontes de aleatoriedade são fornecidas pelo sistema operacional, elas são usadas no lugar da hora do sistema (consulte a função *`os.urandom()`* para detalhes sobre disponibilidade).

Se *a* é um int, ele é usado diretamente.

Com a versão 2 (o padrão), o objeto a *`str`*, *`bytes`* ou *`bytearray`* é convertido em um objeto *`int`* e todos os seus bits são usados.

Com a versão 1 (fornecida para reproduzir sequências aleatórias de versões mais antigas do Python), o algoritmo para *str* e *bytes* gera um intervalo mais restrito de sementes.

Alterado na versão 3.2: Movido para o esquema da versão 2, que usa todos os bits em uma semente de strings.

random.**getstate**()
Retorna um objeto capturando o estado interno atual do gerador. Este objeto pode ser passado para *setstate()* para restaurar o estado.

random.**setstate**(*state*)
state deveria ter sido obtido de uma chamada anterior para *getstate()*, e *setstate()* restaura o estado interno do gerador para o que era no momento *getstate()* foi chamado.

random.**getrandbits**(*k*)
Retorna um número inteiro Python com bits aleatórios *k*. Este método é fornecido com o gerador MersenneTwister e alguns outros geradores também podem fornecê-lo como uma parte opcional da API. Quando disponível, *getrandbits()* permite que *randrange()* manipule intervalos arbitrariamente grandes.

9.6.2 Funções para inteiros

random.**randrange**(*stop*)
random.**randrange**(*start*, *stop*[, *step*])
Retorna um elemento selecionado aleatoriamente em `range(start, stop, step)`. Isso é equivalente a `choice(range(start, stop, step))`, mas na verdade não cria um objeto range.

O padrão de argumento posicional corresponde ao de *range()*. Os argumentos nomeados não devem ser usados porque a função pode usá-los de maneiras inesperadas.

Alterado na versão 3.2: *randrange()* é mais sofisticado em produzir valores igualmente distribuídos. Anteriormente, usava um estilo como `int(random()*n)`, que poderia produzir distribuições ligeiramente desiguais.

random.**randint**(*a*, *b*)
Retorna um inteiro aleatório *N* de forma que `a <= N <= b`. Apelido para `randrange(a, b+1)`.

9.6.3 Funções para sequências

random.**choice**(*seq*)
Retorna um elemento aleatório da sequência não vazia *seq*. Se *seq* estiver vazio, levanta *IndexError*.

random.**choices**(*population*, *weights=None*, *, *cum_weights=None*, *k=1*)
Retorna uma lista do tamanho de *k* de elementos escolhidos da *population* com substituição. Se a *population* estiver vazio, levanta *IndexError*.

Se uma sequência *weights* for especificada, as seleções serão feitas de acordo com os pesos relativos. Alternativamente, se uma sequência *cum_weights* for fornecida, as seleções serão feitas de acordo com os pesos cumulativos (talvez calculados usando *itertools.accumulate()*). Por exemplo, os pesos relativos `[10, 5, 30, 5]` são equivalentes aos pesos acumulados `[10, 15, 45, 50]`. Internamente, os pesos relativos são convertidos em pesos acumulados antes de fazer seleções, portanto, fornecer pesos cumulativos economiza trabalho.

Se nem *weights* nem *cum_weights* forem especificados, as seleções serão feitas com igual probabilidade. Se uma sequência de pesos for fornecida, ela deverá ter o mesmo comprimento que a sequência *population*. É um *TypeError* para especificar ambos os *weights* e *cum_weights*.

The *weights* or *cum_weights* can use any numeric type that interoperates with the *float* values returned by *random()* (that includes integers, floats, and fractions but excludes decimals).

Para uma dada semente, a função *choices()* com igual peso normalmente produz uma sequência diferente das chamadas repetidas para *choice()*. O algoritmo usado por *choice()* usa aritmética de ponto flutuante para

consistência e velocidade internas. O algoritmo usado por *choice()* assume como padrão aritmética inteira com seleções repetidas para evitar pequenos vieses de erro de arredondamento.

Novo na versão 3.6.

random.**shuffle**(*x*[, *random*])
Embaralha a sequência *x* no lugar.

O argumento opcional *random* é uma função de 0 argumentos retornando um ponto flutuante aleatório em [0.0, 1.0); por padrão, esta é a função *random()*.

Para embaralhar uma sequência imutável e retornar uma nova lista embaralhada, use `sample(x, k=len(x))`.

Observe que, mesmo para pequenos `len(x)`, o número total de permutações de *x* pode crescer rapidamente maior que o período da maioria dos geradores de números aleatórios. Isso implica que a maioria das permutações de uma longa sequência nunca pode ser gerada. Por exemplo, uma sequência de comprimento 2080 é a maior que pode caber no período do gerador de números aleatórios Mersenne Twister.

random.**sample**(*population*, *k*)
Retorna uma lista de comprimento *k* de elementos exclusivos escolhidos a partir da sequência ou conjunto da população. Usado para amostragem aleatória sem substituição.

Retorna uma nova lista contendo elementos da população, mantendo a população original inalterada. A lista resultante está na ordem de seleção, para que todas as sub-fatias também sejam amostras aleatórias válidas. Isso permite que os vencedores do sorteio (a amostra) sejam divididos em grandes prêmios e vencedores de segundo lugar (as sub-fatias).

Os membros da população não precisam ser *hasheáveis* ou único. Se a população contiver repetições, cada ocorrência é uma seleção possível na amostra.

Para escolher uma amostra de um intervalo de números inteiros, use um objeto *range()* como argumento. Isso é especialmente rápido e com eficiência de espaço para amostragem de uma grande população: `sample(range(10000000), k=60)`.

Se o tamanho da amostra for maior que o tamanho da população, uma *ValueError* é levantada.

9.6.4 Distribuições com valor real

As funções a seguir geram distribuições específicas com valor real. Os parâmetros de função são nomeados após as variáveis correspondentes na equação da distribuição, conforme usadas na prática matemática comum; a maioria dessas equações pode ser encontrada em qualquer texto estatístico.

random.**random**()
Retorna o próximo número de ponto flutuante aleatório no intervalo [0.0, 1.0).

random.**uniform**(*a*, *b*)
Retorna um número de ponto flutuante aleatório *N* de forma que `a <= N <= b` para `a <= b` e `b <= N <= a` para `b < a`.

O valor do ponto final `b` pode ou não ser incluído no intervalo, dependendo do arredondamento do ponto flutuante na equação `a + (b-a) * random()`.

random.**triangular**(*low*, *high*, *mode*)
Retorna um número de ponto flutuante aleatório *N* de forma que `low <= N <= high` e com o modo *mode* especificado entre esses limites. Os limites *low* e *high* são padronizados como zero e um. O argumento *mode* assume como padrão o ponto médio entre os limites, fornecendo uma distribuição simétrica.

random.**betavariate**(*alpha*, *beta*)
Distribuição beta. As condições nos parâmetros são `alpha > 0` e `beta > 0`. Os valores retornados variam entre 0 e 1.

random.**expovariate**(*lambd*)
Distribuição exponencial. *lambd* é 1.0 dividido pela média desejada. Deve ser diferente de zero. (O parâmetro seria chamado "lambda", mas é uma palavra reservada em Python.) Os valores retornados variam de 0 a infinito positivo se *lambd* for positivo e de infinito negativo a 0 se *lambd* for negativo.

random.**gammavariate**(*alpha*, *beta*)
Distribuição gama. (*Não* a função gama!) As condições nos parâmetros são `alpha > 0` e `beta > 0`.

A função de distribuição de probabilidade é:

```
            x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

random.**gauss**(*mu*, *sigma*)
Distribuição gaussiana. *mu* é a média e *sigma* é o desvio padrão. Isso é um pouco mais rápido que a função *normalvariate()* definida abaixo.

random.**lognormvariate**(*mu*, *sigma*)
Distribuição log normal. Se você usar o logaritmo natural dessa distribuição, obterá uma distribuição normal com média *mu* e desvio padrão *sigma*. *mu* pode ter qualquer valor e *sigma* deve ser maior que zero.

random.**normalvariate**(*mu*, *sigma*)
Distribuição normal. *mu* é a média e *sigma* é o desvio padrão.

random.**vonmisesvariate**(*mu*, *kappa*)
mu é o ângulo médio, expresso em radianos entre 0 e 2**pi*, e *kappa* é o parâmetro de concentração, que deve ser maior ou igual a zero. Se *kappa* for igual a zero, essa distribuição será reduzida para um ângulo aleatório uniforme no intervalo de 0 a 2**pi*.

random.**paretovariate**(*alpha*)
Distribuição de Pareto. *alpha* é o parâmetro de forma.

random.**weibullvariate**(*alpha*, *beta*)
Distribuição Weibull. *alpha* é o parâmetro de escala e *beta* é o parâmetro de forma.

9.6.5 Gerador alternativo

class random.**Random**([*seed*])
Classe que implementa o gerador de números pseudoaleatórios padrão usado pelo módulo *random*.

class random.**SystemRandom**([*seed*])
Classe que usa a função *os.urandom()* para gerar números aleatórios a partir de fontes fornecidas pelo sistema operacional. Não disponível em todos os sistemas. Não depende do estado do software e as sequências não são reproduzíveis. Assim, o método *seed()* não tem efeito e é ignorado. Os métodos *getstate()* e *setstate()* levantam *NotImplementedError* se chamados.

9.6.6 Notas sobre reprodutibilidade

Sometimes it is useful to be able to reproduce the sequences given by a pseudo random number generator. By re-using a seed value, the same sequence should be reproducible from run to run as long as multiple threads are not running.

A maioria dos algoritmos e funções de propagação do módulo aleatório está sujeita a alterações nas versões do Python, mas dois aspectos são garantidos para não serem alterados:

- Se um novo método de semeadura for adicionado, será oferecida uma semeadora compatível com versões anteriores.
- O método do gerador `random()` continuará produzindo a mesma sequência quando o semeador compatível receber a mesma semente.

9.6.7 Exemplos e receitas

Exemplos básicos:

```
>>> random()                            # Random float:  0.0 <= x < 1.0
0.37444887175646646

>>> uniform(2.5, 10.0)                  # Random float:  2.5 <= x < 10.0
3.1800146073117523

>>> expovariate(1 / 5)                  # Interval between arrivals averaging 5␣
↪seconds
5.148957571865031

>>> randrange(10)                       # Integer from 0 to 9 inclusive
7

>>> randrange(0, 101, 2)                # Even integer from 0 to 100 inclusive
26

>>> choice(['win', 'lose', 'draw'])     # Single random element from a sequence
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                       # Shuffle a list
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4)   # Four samples without replacement
[40, 10, 50, 30]
```

Simulações:

```
>>> # Six roulette wheel spins (weighted sampling with replacement)
>>> choices(['red', 'black', 'green'], [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']

>>> # Deal 20 cards without replacement from a deck of 52 playing cards
>>> # and determine the proportion of cards with a ten-value
>>> # (a ten, jack, queen, or king).
>>> deck = collections.Counter(tens=16, low_cards=36)
>>> seen = sample(list(deck.elements()), k=20)
>>> seen.count('tens') / 20
0.15
```

(continua na próxima página)

(continuação da página anterior)

```
>>> # Estimate the probability of getting 5 or more heads from 7 spins
>>> # of a biased coin that settles on heads 60% of the time.
>>> def trial():
...     return choices('HT', cum_weights=(0.60, 1.00), k=7).count('H') >= 5
...
>>> sum(trial() for i in range(10000)) / 10000
0.4169

>>> # Probability of the median of 5 samples being in middle two quartiles
>>> def trial():
...     return 2500 <= sorted(choices(range(10000), k=5))[2] < 7500
...
>>> sum(trial() for i in range(10000)) / 10000
0.7958
```

Example of statistical bootstrapping using resampling with replacement to estimate a confidence interval for the mean of a sample of size five:

```
# http://statistics.about.com/od/Applications/a/Example-Of-Bootstrapping.htm
from statistics import mean
from random import choices

data = 1, 2, 4, 4, 10
means = sorted(mean(choices(data, k=5)) for i in range(20))
print(f'The sample mean of {mean(data):.1f} has a 90% confidence '
      f'interval from {means[1]:.1f} to {means[-2]:.1f}')
```

Exemplo de um teste de permutação de reamostragem pára determinar a significância estatística ou valor-p de uma diferença observada entre os efeitos de uma droga em comparação com um placebo:

```
# Example from "Statistics is Easy" by Dennis Shasha and Manda Wilson
from statistics import mean
from random import shuffle

drug = [54, 73, 53, 70, 73, 68, 52, 65, 65]
placebo = [54, 51, 58, 44, 55, 52, 42, 47, 58, 46]
observed_diff = mean(drug) - mean(placebo)

n = 10000
count = 0
combined = drug + placebo
for i in range(n):
    shuffle(combined)
    new_diff = mean(combined[:len(drug)]) - mean(combined[len(drug):])
    count += (new_diff >= observed_diff)

print(f'{n} label reshufflings produced only {count} instances with a difference')
print(f'at least as extreme as the observed difference of {observed_diff:.1f}.')
print(f'The one-sided p-value of {count / n:.4f} leads us to reject the null')
print(f'hypothesis that there is no difference between the drug and the placebo.')
```

Simulation of arrival times and service deliveries in a single server queue:

```
from random import expovariate, gauss
from statistics import mean, median, stdev
```

(continua na próxima página)

(continuação da página anterior)

```
average_arrival_interval = 5.6
average_service_time = 5.0
stdev_service_time = 0.5

num_waiting = 0
arrivals = []
starts = []
arrival = service_end = 0.0
for i in range(20000):
    if arrival <= service_end:
        num_waiting += 1
        arrival += expovariate(1.0 / average_arrival_interval)
        arrivals.append(arrival)
    else:
        num_waiting -= 1
        service_start = service_end if num_waiting else arrival
        service_time = gauss(average_service_time, stdev_service_time)
        service_end = service_start + service_time
        starts.append(service_start)

waits = [start - arrival for arrival, start in zip(arrivals, starts)]
print(f'Mean wait: {mean(waits):.1f}.  Stdev wait: {stdev(waits):.1f}.')
print(f'Median wait: {median(waits):.1f}.  Max wait: {max(waits):.1f}.')
```

Ver também:

Statistics for Hackers um tutorial em vídeo por Jake Vanderplas sobre análise estatística usando apenas alguns conceitos fundamentais, incluindo simulação, amostragem, embaralhamento e validação cruzada.

Economics Simulation uma simulação de um mercado por Peter Norvig isso mostra o uso eficaz de muitas das ferramentas e distribuições fornecidas por este módulo (gauss, uniforme, amostra, betavariado, escolha, triangular e intervalo).

A Concrete Introduction to Probability (using Python) um tutoria por Peter Norvig cobrindo o básico da teoria das probabilidades, como escrever simulações e como realizar análise de dados usando Python.

9.7 `statistics` — Funções estatísticas

Novo na versão 3.4.

Código Fonte: Lib/statistics.py

Este módulo fornece funções para calcular estatísticas de dados numéricos (com valor `Real`)

Nota: A não ser que marcado explicitamente o contrário, essas funções suportam valores dos tipos *`int`*, *`float`*, *`decimal.Decimal`* e *`fractions.Fraction`*. O uso com outros tipos (independente de serem numéricos ou não) ainda não é suportado. O mesmo vale para tipos mistos, cujo comportamento dependerá da implementação. Se os seus dados de entrada possuem diversos tipos, talvez você consiga usar da função *`map()`* para garantir resultados consistentes. Por exemplo `map(float, input_data)` converterá os elementos de `input_data` para float.

9.7.1 Médias e medidas de valor central

Essas funções calculam a média ou o valor típico de uma população ou amostra.

mean()	Média aritmética dos dados.
harmonic_mean()	Média harmônica dos dados.
median()	Mediana (valor do meio) dos dados.
median_low()	Mediana inferior dos dados.
median_high()	Mediana superior dos dados.
median_grouped()	Mediana, ou o 50º percentil dos dados agrupados.
mode()	Moda (valor mais comum) de dados discretos.

9.7.2 Medidas de espalhamento

Essas funções calculam o quanto a população ou amostra tendem a desviar dos valores típicos ou médios.

pstdev()	Desvio padrão de dados populacionais.
pvariance()	Variância de dados populacionais.
stdev()	Desvio padrão de dados amostrais.
variance()	Variância de dados amostrais.

9.7.3 Detalhes da função

Nota: as funções não exigem que os dados estejam ordenados. No entanto, para conveniência do leitor, a maioria dos exemplos mostrará sequências ordenadas.

`statistics.`**`mean`**`(`*data*`)`

Retorna a média aritmética de *data*, que pode ser uma sequência ou um iterador

A média aritmética é a soma dos dados dividida pela quantidade. É comumente chamada apenas de "média", apesar de ser uma das diversas médias matemáticas. Ela representa uma medida da localização central dos dados

Se *data* for vazio, uma exceção do tipo *StatisticsError* será lançada.

Alguns exemplos de uso:

```
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

Nota: A média é bastante afetada por valores discrepantes e não é uma estimativa muito robusta da localização central. Ou seja, a média não é necessariamente um ponto que ocorre nos dados. Para medidas mais robustas, apesar de menos eficientes, da localização central, veja *median()* e *mode()*. Nesse caso "eficiência" se refere à eficiência estatística ao invés de computacional.

A média de uma amostra nos dá uma estimativa não enviesada da média populacional verdadeira. Isso significa que obtendo a média de todas as amostras, `mean(sample)` converge para a média da população como um todo. Se *data* representa a população inteira ao invés de uma amostra, então `mean(data)` é equivalente a calcular a média verdadeira da população, comumente denotada por μ.

`statistics.`**`harmonic_mean`**`(`*data*`)`

Retorna a média harmônica de *data*, que deve ser uma sequência ou um iterador de números reais.

A média harmônica, as vezes chamada de média subcontrária é a recíproca da média aritmética, calculada por *mean()*, dos recíprocos dos dados. Por exemplo, a média harmônica dos valores *a*, *b* e *c* será equivalente a `3/(1/a + 1/b + 1/c)`.

A média harmônica, como toda média, é uma medida da localização central dos dados. Ela costuma ser apropriada quanto estamos tirando média de quantidades que representam taxas ou razões, como velocidade. Por exemplo:

Suponha que um investidor compre um valor igual de ações em cada uma de três companhias diferentes com uma razão P/L (preço/lucro) de 2,5, 3 e 10. Qual é a razão P/L média da carteira de investimentos desse investidor?

```
>>> harmonic_mean([2.5, 3, 10])  # For an equal investment portfolio.
3.6
```

Usando a média aritmética, teríamos um valor de aproximadamente 5.167, que é muito alto.

StatisticsError é levantado se *data* for vazio ou se qualquer elemento for menor que zero.

Novo na versão 3.6.

`statistics.`**`median`**`(`*data*`)`

Retorna a mediana (o valor do meio) de dados numéricos, usando o método comum de "média entre os dois do meio". Se *data* for vazio, *StatisticsError* é levantado. *data* pode ser uma sequência ou um iterador.

A mediana é uma medida robusta da localização central e é menos afetada pela presença de valores muito discrepantes nos seus dados. Quando o número de elementos nos dados for ímpar, o valor que fica no meio da sequência é retornado:

```
>>> median([1, 3, 5])
3
```

Quando o número de elementos for par, a mediana é calculada tomando a média entre os dois valores no meio:

```
>>> median([1, 3, 5, 7])
4.0
```

Isso serve quando seus dados forem discretos e você não se importa que a média possa não ser um valor que de fato ocorre nos seus dados.

Caso os seus dados sejam ordinais (ou seja, suportam operações de ordenação) mas não são numéricos (não suportam adição), você deve usar *median_low()* ou *median_high()* no lugar.

Ver também:

median_low(), *median_high()*, *median_grouped()*

`statistics.`**`median_low`**`(`*data*`)`

Retorna a mediana inferior de dados numéricos. Se *data* for vazio, *StatisticsError* é levantado. *data* pode ser uma sequência ou um iterador.

A mediana inferior sempre é um membro do conjunto de dados. Quando o número de elementos for ímpar, o valor intermediário é retornado. Se houver um número par de elementos, o menor entre os dois valores centrais é retornado.

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

Use *median_low()* quando seus dados são discretos e você prefere que a mediana seja um valor que de fato existe nos seus dados ao invés de um valor interpolado.

statistics.**median_high**(*data*)

Retorna a mediana superior de dados numéricos. Se *data* for vazio, *StatisticsError* é levantado. *data* pode ser uma sequência ou um iterador.

A mediana superior sempre é um membro do conjunto de dados. Quando o número de elementos for ímpar, o valor intermediário é retornado. Se houver um número par de elementos, o maior entre os dois valores centrais é retornado.

```
>>> median_high([1, 3, 5])
3
>>> median_high([1, 3, 5, 7])
5
```

Use *median_high()* quando seus dados são discretos e você prefere que a mediana seja um valor que de fato existe nos seus dados ao invés de um valor interpolado.

statistics.**median_grouped**(*data*, *interval=1*)

Retorna a media de dados contínuos agrupados, calculada como o 50º percentil, usando de interpolação. Se *data* for vazio, *StatisticsError* é levantado. *data* pode ser uma sequência ou um iterador.

```
>>> median_grouped([52, 52, 53, 54])
52.5
```

No exemplo a seguir, os dados estão arredondados de forma que cada valor representa o ponto intermediário de classes de dados. Isto é, 1 é o meio da classe 0.5–1.5, 2 é o meio de 1.5–2.5, 3 é o meio de 2.5–3.5, etc. Com os dados oferecidos, o valor do meio cai em algum ponto na classe 3.5–4.5 e interpolação é usada para estimá-lo:

```
>>> median_grouped([1, 2, 2, 3, 4, 4, 4, 4, 4, 5])
3.7
```

O argumento opcional *interval* representa a classe de intervalo e tem como valor padrão 1. Mudar a classe de intervalo irá mudar a interpolação:

```
>>> median_grouped([1, 3, 3, 5, 7], interval=1)
3.25
>>> median_grouped([1, 3, 3, 5, 7], interval=2)
3.5
```

Essa função não checa se os pontos de dados estão separados por uma distância que seja ao menos igual à *interval*

CPython implementation detail: Em algumas circunstâncias *median_grouped()* pode forçar os pontos de dados a serem floats. Esse comportamento provavelmente irá mudar no futuro.

Ver também:

- "Statistics for the Behavioral Sciences", Frederick J Gravetter and Larry B Wallnau (8th Edition).
- A função SSMEDIAN <https://help.gnome.org/users/gnumeric/stable/gnumeric.html#gnumeric-function-SSMEDIAN>'_ na planilha Gnome Gnumeric, incluindo essa discussão <https://mail.gnome.org/archives/gnumeric-list/2011-April/msg00018.html>'_.

statistics.**mode**(*data*)

Retorna o valor mais comum em dados discretos ou nominais. A moda - quando ela existe - é o valor mais típico e é uma medida robusta da localização central

Se *data* for vazio ou se não tem exatamente um valor mais comum, *StatisticsError* é levantado.

`mode` assume que os dados são discretos e retorna um único valor. Esse é o tratamento padrão do conceito de moda normalmente ensinado nas escolas:

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

A moda é única no sentido que é a única medida estatística que também se aplica a dados nominais (não numéricos):

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

statistics.**pstdev**(*data*, *mu=None*)

Retorna o desvio padrão da população. Ou seja, a raíz quadrada da variância da população. Veja os argumentos e outros detalhes em *pvariance()*.

```
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

statistics.**pvariance**(*data*, *mu=None*)

Retorna a variância populacional de *data*, que deve ser um iterável não vazio de números reais. A variância, o segundo Momento Estatístico sobre a média é uma medida da variabilidade (espalhamento ou dispersão) dos dados. Uma variância grande indica que os dados são espalhados; uma variância menor indica que os dados estão agrupado em volta da média.

Se o segundo argumento opcional *mu* for dado, ele deve representar a média de *data*. Se ele não estiver presente ou for `None`, a média será automaticamente calculada.

Use essa função para calcular a variância de toda a população. Para estimar a variância de uma amostra, a função *variance()* costuma ser uma escolha melhor.

Causa *StatisticsError* se *data* for vazio.

Exemplos:

```
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.75, 3.25]
>>> pvariance(data)
1.25
```

Se você já calculou a média dos seus dados, você pode passar o valor no segundo argumento opcional *mu* para evitar recálculos:

```
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```

Essa função não tentar verificar que você passou a média verdadeira em *mu*. Se você passar um valor arbitrário para *mu*, poderá ter resultados inválidos ou impossíveis.

Decimais e frações são suportadas:

```
>>> from decimal import Decimal as D
>>> pvariance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('24.815')
```

(continua na próxima página)

(continuação da página anterior)

```
>>> from fractions import Fraction as F
>>> pvariance([F(1, 4), F(5, 4), F(1, 2)])
Fraction(13, 72)
```

Nota: Quando os dados de entrada representarem toda a população, ele retorna a variância populacional σ^2. Se em vez disso, amostras forem usadas, então a variância amostral enviesada s^2, também conhecida como variância com N graus de liberdade é retornada.

If you somehow know the true population mean μ, you may use this function to calculate the variance of a sample, giving the known population mean as the second argument. Provided the data points are representative (e.g. independent and identically distributed), the result will be an unbiased estimate of the population variance.

`statistics.`**`stdev`**(*data*, *xbar=None*)

Retorna o desvio padrão amostral (a raiz quadrada da variância da amostra). Veja *`variance()`* para argumentos e outros detalhes.

```
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```

`statistics.`**`variance`**(*data*, *xbar=None*)

Retorna a variância amostral de *data*, que deve ser um iterável com pelo menos dois números reais. Variância, ou o segundo momento estatístico sobre a média, é uma medida de variabilidade (espalhamento ou dispersão) dos dados. Uma variância grande indica que os dados são espalhados, uma variância menor indica que os dados estão agrupados em volta da média.

Se o segundo argumento opcional *xbar* for dado, ele deve representar a média de *data*. Se ele não estiver presente ou for `None` (valor padrão), a média é automaticamente calculada.

Use essa função quando seus dados representarem uma amostra da população. Para calcular a variância de toda população veja *`pvariance()`*.

Levanta a exceção *`StatisticsError`* se *data* tiver menos do que dois valores.

Exemplos:

```
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

Se você já calculou a média dos seus dados, você pode passar o valor no segundo argumento opcional *xbar* para evitar recálculos:

```
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

Essa função não verifica se você passou a média verdadeira como *xbar*. Usando valores arbitrários para *xbar* pode levar a resultados inválidos ou impossíveis.

Valores decimais e fracionários são suportados.

```
>>> from decimal import Decimal as D
>>> variance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('31.01875')
```

(continua na próxima página)

(continuação da página anterior)

```
>>> from fractions import Fraction as F
>>> variance([F(1, 6), F(1, 2), F(5, 3)])
Fraction(67, 108)
```

Nota: Essa é a variância amostral s^2 com a correção de Bessel, também conhecida como variância com N-1 graus de liberdade. Desde que os pontos de dados sejam representativos (por exemplo, independentes e distribuídos de forma idêntica), o resultado deve ser uma estimativa não enviesada da verdadeira variação populacional.

Caso você de alguma forma saiba a verdadeira média populacional μ você deveria passar para a função *pvariance()* como o parâmetro *mu* para pegar a variância da amostra.

9.7.4 Exceções

Uma única exceção é definida:

exception statistics.**StatisticsError**
Subclasse de *ValueError* para exceções relacionadas a estatísticas.

CAPÍTULO 10

Módulos de Programção Funcional

Os módulos descritos neste capítulo fornecem funções e classes que suportam um estilo de programação funcional e operações gerais em callables.

Os seguintes módulos estão documentados neste capítulo:

10.1 `itertools` — Funções que criam iteradores para laços eficientes

Esse módulo implementa diversos blocos de construção de *iterator* building blocks inspirados por construções de APL, Haskell, e SML. Cada uma foi reformulada de forma adequada para Python.

Esse módulo padroniza um conjunto central de ferramentas rápidas e de uso eficiente da memória, que podem ser utilizadas sozinhas ou combinadas. Juntas, eles formam uma "álgebra de iteradores" tornando possível construir ferramentas sucintas e eficientes em Python puro.

Por exemplo, SML fornece uma ferramenta para tabulação: `tabulate(f)` que produz uma sequência `f(0), f(1), ...`. O mesmo efeito pode ser obtido em Python combinando *`map()`* e *`count()`* para formar `map(f, count())`.

Essa ferramentas e suas equivalências embutidas também trabalham bem com as funções de alta velocidade do módulo *`operator`* module. Por exemplo, o operador de multiplicação pode ser mapeado em dois vetores para criar um produto escalar eficiente: `sum(map(operator.mul, vector1, vector2))`.

Iteradores infinitos:

Iterador	Argumentos	Resultado	Exemplo
count()	start, [step]	start, start+step, start+2*step, …	`count(10) --> 10 11 12 13 14 ...`
cycle()	p	p0, p1, … ultimo elemento de p, p0, p1, …	`cycle('ABCD') --> A B C D A B C D ...`
repeat()	elem [,n]	elem, elem, elem, … repete infinitamente ou até n vezes	`repeat(10, 3) --> 10 10 10`

Iteradores terminando na sequencia de entrada mais curta:

Iterador	Argumentos	Resultado	Exemplo
accumulate()	p [,func]	p0, p0+p1, p0+p1+p2, …	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`
chain()	p, q, …	p0, p1, … último elemento de p, q0, q1, …	`chain('ABC', 'DEF') --> A B C D E F`
chain.from_iterable()	iterável	p0, p1, … último elemento de p, q0, q1, …	`chain.from_iterable(['ABC', 'DEF']) --> A B C D E F`
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), …	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`
dropwhile()	pred, seq	seq[n], seq[n+1], iniciando quando pred for falsa	`dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1`
filterfalse()	pred, seq	elementos de seq onde pred(elem) é falso	`filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8`
groupby()	iterable[, key]	sub-iteradores agrupados pelo valor de key(v)	
islice()	seq, [start,] stop [, step]	elementos de seq[start:stop:step]	`islice('ABCDEFG', 2, None) --> C D E F G`
starmap()	func, seq	func(*seq[0]), func(*seq[1]), …	`starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000`
takewhile()	pred, seq	seq[0], seq[1], enquanto pred é falso	`takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4`
tee()	it, n	n iteradores it independentes	
zip_longest()	p, q, …	(p[0], q[0]), (p[1], q[1]), …	`zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-`

Iteradores combinatórios:

Iterador	Argumentos	Resultado
product()	p, q, … [repeat=1]	produto cartesiano, equivalente a laços `for` aninhados
permutations()	p[, r]	tuplas de tamanho r, com todas ordenações possíveis, sem elementos repetidos
combinations()	p, r	tuplas de tamanho r, ordenadas, sem elementos repetidos
combinations_with_replacement()	p, r	tuplas de tamanho r, ordenadas, com elementos repetidos
`product('ABCD', repeat=2)`		`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`		`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`		`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`		`AA AB AC AD BB BC BD CC CD DD`

10.1.1 Funções de itertools

Todas as funções a seguir constroem e retorna iteradores. Algumas fornecem fluxos de tamanhos infinitos, assim elas devem ser acessados somente por funções ou laços que interrompem o fluxo.

`itertools.`**`accumulate`**(*iterable*[, *func*])

Make an iterator that returns accumulated sums, or accumulated results of other binary functions (specified via the optional *func* argument). If *func* is supplied, it should be a function of two arguments. Elements of the input *iterable* may be any type that can be accepted as arguments to *func*. (For example, with the default operation of addition, elements may be any addable type including *Decimal* or *Fraction*.) If the input iterable is empty, the output iterable will also be empty.

Aproximadamente equivalente a:

```
def accumulate(iterable, func=operator.add):
    'Return running totals'
    # accumulate([1,2,3,4,5]) --> 1 3 6 10 15
    # accumulate([1,2,3,4,5], operator.mul) --> 1 2 6 24 120
    it = iter(iterable)
    try:
        total = next(it)
    except StopIteration:
        return
    yield total
    for element in it:
        total = func(total, element)
        yield total
```

Existem diversos usos para o argumento *func*. Ele pode ser definido como a função *min()* calcular um valor mínimo, *max()* para um valor máximo, ou *operator.mul()* para calcular um produto. Tabelas de amortização podem ser construídas acumulando juros e aplicando pagamentos. Relações de recorrência de primeira ordem podem ser modeladas fornecendo o valor inicial no iterável e usando somente o valor total acumulado no argumento *func*:

```
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, operator.mul))     # running product
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]
```

(continua na próxima página)

(continuação da página anterior)

```
>>> list(accumulate(data, max))                # running maximum
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]

# Amortize a 5% loan of 1000 with 4 annual payments of 90
>>> cashflows = [1000, -90, -90, -90, -90]
>>> list(accumulate(cashflows, lambda bal, pmt: bal*1.05 + pmt))
[1000, 960.0, 918.0, 873.9000000000001, 827.5950000000001]

# Chaotic recurrence relation https://en.wikipedia.org/wiki/Logistic_map
>>> logistic_map = lambda x, _:  r * x * (1 - x)
>>> r = 3.8
>>> x0 = 0.4
>>> inputs = repeat(x0, 36)     # only the initial value is used
>>> [format(x, '.2f') for x in accumulate(inputs, logistic_map)]
['0.40', '0.91', '0.30', '0.81', '0.60', '0.92', '0.29', '0.79', '0.63',
 '0.88', '0.39', '0.90', '0.33', '0.84', '0.52', '0.95', '0.18', '0.57',
 '0.93', '0.25', '0.71', '0.79', '0.63', '0.88', '0.39', '0.91', '0.32',
 '0.83', '0.54', '0.95', '0.20', '0.60', '0.91', '0.30', '0.80', '0.60']
```

Veja *functools.reduce()* para uma função similar que devolve apenas o valor acumulado final.

Novo na versão 3.2.

Alterado na versão 3.3: Adicionado o parâmetro opcional *func*.

itertools.**chain**(**iterables*)

Cria um iterador que devolve elementos do primeiro iterável até o esgotamento, então continua com o próximo iterável, até que todos os iteráveis sejam esgotados. Usando para tratar sequências consecutivas como uma única sequencia. aproximadamente equivalente a:

```
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

classmethod chain.**from_iterable**(*iterable*)

Construtor alternativo para *chain()*. Obtém entradas encadeadas a partir de um único argumento iterável que avaliado preguiçosamente. Aproximadamente equivalente a:

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

itertools.**combinations**(*iterable*, *r*)

Devolve subsequências de elementos com comprimento *r* a partir da entrada *iterável*

Combinations are emitted in lexicographic sort order. So, if the input *iterable* is sorted, the combination tuples will be produced in sorted order.

Os elementos são tratados como únicos baseado em suas posições, não em seus valores. Portanto se os elementos de entrada são únicos, não haverá repetição de valores nas sucessivas combinações.

Aproximadamente equivalente a:

```
def combinations(iterable, r):
    # combinations('ABCD', 2) --> AB AC AD BC BD CD
    # combinations(range(4), 3) --> 012 013 023 123
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = list(range(r))
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

O código para *combinations()* também pode ser expresso como uma subsequência de *permutations()* depois de filtradas as entradas onde os elementos não estão ordenandos (de acordo com a sua posição na entrada):

```
def combinations(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in permutations(range(n), r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

O número de itens devolvidos é `n! / r! / (n-r)!` quando `0 <= r <= n` ou zero quando `r > n`.

itertools.**combinations_with_replacement**(*iterable*, *r*)

Devolve subsequências de comprimento *r* de elementos do *iterável* de entrada permitindo que elementos individuais sejam repetidos mais de uma vez.

Combinations are emitted in lexicographic sort order. So, if the input *iterable* is sorted, the combination tuples will be produced in sorted order.

Os elementos são tratados como únicos baseado em suas posições, não em seus valores. Portanto se os elementos de entrada forem únicos, não haverá repetição de valores nas combinações geradas.

Aproximadamente equivalente a:

```
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) --> AA AB AC BB BC CC
    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
        return
    indices = [0] * r
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
```

(continua na próxima página)

(continuação da página anterior)

```
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

O código para *combinations_with_replacement()* também pode ser expresso como uma subsequência de *product()* depois de filtradas as entradas onde os elementos não estão ordenados (de acordo com a sua posição na entrada):

```
def combinations_with_replacement(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in product(range(n), repeat=r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

O número de itens devolvidos é `(n+r-1)! / r! / (n-1)!` quando `n > 0`.

Novo na versão 3.1.

`itertools.`**`compress`**(*data*, *selectors*)

Crie um iterador que filtra elementos de *data* devolvendo apenas aqueles que tem um elemento correspondente em *selectors* que seja avaliado `True`. Interrompe quando os iteráveis *data* ou *selectors* tiveram sido esgotados. Aproximadamente equivalente a:

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
    return (d for d, s in zip(data, selectors) if s)
```

Novo na versão 3.1.

`itertools.`**`count`**(*start=0*, *step=1*)

Crie um iterador que devolve valores igualmente espaçados começando pelo número *start*. Frequentemente usado com um argumento da função *map()* para gerar pontos de dados consecutivos. Também usado com *zip()* para adicionar números sequenciais. Aproximadamente equivalente a:

```
def count(start=0, step=1):
    # count(10) --> 10 11 12 13 14 ...
    # count(2.5, 0.5) -> 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

Quando é feita uma contagem usando números de ponto flutuante, é possível ter melhor precisão substituindo código multiplicativo como `(start + step * i for i in count())`.

Alterado na versão 3.1: Adicionou argumento *step* e permitiu argumentos não-inteiros.

`itertools.`**`cycle`**(*iterable*)

Crie um iterador que devolve elementos do iterável assim como salva uma cópia de cada um. Quando o iterável é esgotado, devolve elementos da cópia salva. Repete indefinidamente. Aproximadamente equivalente a:

```
def cycle(iterable):
    # cycle('ABCD') --> A B C D A B C D A B C D ...
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
```

(continua na próxima página)

(continuação da página anterior)

```
    while saved:
        for element in saved:
              yield element
```

Note, this member of the toolkit may require significant auxiliary storage (depending on the length of the iterable).

itertools.**dropwhile**(*predicate*, *iterable*)

Make an iterator that drops elements from the iterable as long as the predicate is true; afterwards, returns every element. Note, the iterator does not produce *any* output until the predicate first becomes false, so it may have a lengthy start-up time. Roughly equivalent to:

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

itertools.**filterfalse**(*predicate*, *iterable*)

Make an iterator that filters elements from iterable returning only those for which the predicate is `False`. If *predicate* is `None`, return the items that are false. Roughly equivalent to:

```
def filterfalse(predicate, iterable):
    # filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```

itertools.**groupby**(*iterable*, *key=None*)

Make an iterator that returns consecutive keys and groups from the *iterable*. The *key* is a function computing a key value for each element. If not specified or is `None`, *key* defaults to an identity function and returns the element unchanged. Generally, the iterable needs to already be sorted on the same key function.

The operation of *groupby()* is similar to the `uniq` filter in Unix. It generates a break or new group every time the value of the key function changes (which is why it is usually necessary to have sorted the data using the same key function). That behavior differs from SQL's GROUP BY which aggregates common elements regardless of their input order.

The returned group is itself an iterator that shares the underlying iterable with *groupby()*. Because the source is shared, when the *groupby()* object is advanced, the previous group is no longer visible. So, if that data is needed later, it should be stored as a list:

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # Store group iterator as a list
    uniquekeys.append(k)
```

groupby() é aproximadamente equivalente a:

```
class groupby:
    # [k for k, g in groupby('AAAABBBCCDAABBB')] --> A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] --> AAAA BBB CC D
    def __init__(self, iterable, key=None):
        if key is None:
            key = lambda x: x
        self.keyfunc = key
        self.it = iter(iterable)
        self.tgtkey = self.currkey = self.currvalue = object()
    def __iter__(self):
        return self
    def __next__(self):
        self.id = object()
        while self.currkey == self.tgtkey:
            self.currvalue = next(self.it)    # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
        self.tgtkey = self.currkey
        return (self.currkey, self._grouper(self.tgtkey, self.id))
    def _grouper(self, tgtkey, id):
        while self.id is id and self.currkey == tgtkey:
            yield self.currvalue
            try:
                self.currvalue = next(self.it)
            except StopIteration:
                return
            self.currkey = self.keyfunc(self.currvalue)
```

itertools.**islice**(*iterable*, *stop*)
itertools.**islice**(*iterable*, *start*, *stop*[, *step*])

Make an iterator that returns selected elements from the iterable. If *start* is non-zero, then elements from the iterable are skipped until start is reached. Afterward, elements are returned consecutively unless *step* is set higher than one which results in items being skipped. If *stop* is `None`, then iteration continues until the iterator is exhausted, if at all; otherwise, it stops at the specified position. Unlike regular slicing, *islice()* does not support negative values for *start*, *stop*, or *step*. Can be used to extract related fields from data where the internal structure has been flattened (for example, a multi-line report may list a name field on every third line). Roughly equivalent to:

```
def islice(iterable, *args):
    # islice('ABCDEFG', 2) --> A B
    # islice('ABCDEFG', 2, 4) --> C D
    # islice('ABCDEFG', 2, None) --> C D E F G
    # islice('ABCDEFG', 0, None, 2) --> A C E G
    s = slice(*args)
    start, stop, step = s.start or 0, s.stop or sys.maxsize, s.step or 1
    it = iter(range(start, stop, step))
    try:
        nexti = next(it)
    except StopIteration:
        # Consume *iterable* up to the *start* position.
        for i, element in zip(range(start), iterable):
            pass
        return
    try:
        for i, element in enumerate(iterable):
            if i == nexti:
                yield element
                nexti = next(it)
    except StopIteration:
```

(continua na próxima página)

(continuação da página anterior)

```
        # Consume to *stop*.
        for i, element in zip(range(i + 1, stop), iterable):
            pass
```

If *start* is `None`, then iteration starts at zero. If *step* is `None`, then the step defaults to one.

`itertools.`**`permutations`**(*iterable*, *r=None*)

Return successive *r* length permutations of elements in the *iterable*.

If *r* is not specified or is `None`, then *r* defaults to the length of the *iterable* and all possible full-length permutations are generated.

Permutations are emitted in lexicographic sort order. So, if the input *iterable* is sorted, the permutation tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each permutation.

Aproximadamente equivalente a:

```
def permutations(iterable, r=None):
    # permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) --> 012 021 102 120 201 210
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return
    indices = list(range(n))
    cycles = list(range(n, n-r, -1))
    yield tuple(pool[i] for i in indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

O código para *permutations()* também pode ser expresso como uma subsequência de *product()* depois de filtradas as entradas com elementos repetidos (os de mesma posição no conjunto de entrada):

```
def permutations(iterable, r=None):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    for indices in product(range(n), repeat=r):
        if len(set(indices)) == r:
            yield tuple(pool[i] for i in indices)
```

O número de itens retornado é `n! / (n-r)!` quando `0 <= r <= n` ou zero quando `r > n`.

itertools.**product**(**iterables*, *repeat=1*)
Produto cartesiano de iteráveis de entrada

Aproximadamente equivalente a laços for aninhados em uma expressão geradora. Por exemplo, `product(A, B)` devolve o mesmo que `((x,y) for x in A for y in B)`.

Os laços aninhados circulam como um hodômetro com o elemento mais à direita avançando a cada iteração. Este padrão cria uma ordenação lexicográfica de maneira que se os iteráveis de entrada estiverem ordenados, as tuplas produzidas são emitidas de maneira ordenada.

To compute the product of an iterable with itself, specify the number of repetitions with the optional *repeat* keyword argument. For example, `product(A, repeat=4)` means the same as `product(A, A, A, A)`.

This function is roughly equivalent to the following code, except that the actual implementation does not build up intermediate results in memory:

```
def product(*args, repeat=1):
    # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) --> 000 001 010 011 100 101 110 111
    pools = [tuple(pool) for pool in args] * repeat
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)
```

itertools.**repeat**(*object*[, *times*])
Make an iterator that returns *object* over and over again. Runs indefinitely unless the *times* argument is specified. Used as argument to *map()* for invariant parameters to the called function. Also used with *zip()* to create an invariant part of a tuple record.

Aproximadamente equivalente a:

```
def repeat(object, times=None):
    # repeat(10, 3) --> 10 10 10
    if times is None:
        while True:
            yield object
    else:
        for i in range(times):
            yield object
```

A common use for *repeat* is to supply a stream of constant values to *map* or *zip*:

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

itertools.**starmap**(*function*, *iterable*)
Make an iterator that computes the function using arguments obtained from the iterable. Used instead of *map()* when argument parameters are already grouped in tuples from a single iterable (the data has been "pre-zipped"). The difference between *map()* and *starmap()* parallels the distinction between `function(a,b)` and `function(*c)`. Roughly equivalent to:

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

itertools.**takewhile**(*predicate*, *iterable*)
Make an iterator that returns elements from the iterable as long as the predicate is true. Roughly equivalent to:

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

itertools.**tee**(*iterable*, *n=2*)
Return *n* independent iterators from a single iterable.

The following Python code helps explain what *tee* does (although the actual implementation is more complex and uses only a single underlying FIFO queue).

Aproximadamente equivalente a:

```
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:             # when the local deque is empty
                try:
                    newval = next(it)   # fetch a new value and
                except StopIteration:
                    return
                for d in deques:        # load it to all the deques
                    d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

Once *tee()* has made a split, the original *iterable* should not be used anywhere else; otherwise, the *iterable* could get advanced without the tee objects being informed.

`tee` iterators are not threadsafe. A *RuntimeError* may be raised when using simultaneously iterators returned by the same *tee()* call, even if the original *iterable* is threadsafe.

This itertool may require significant auxiliary storage (depending on how much temporary data needs to be stored). In general, if one iterator uses most or all of the data before another iterator starts, it is faster to use *list()* instead of *tee()*.

itertools.**zip_longest**(**iterables*, *fillvalue=None*)
Make an iterator that aggregates elements from each of the iterables. If the iterables are of uneven length, missing values are filled-in with *fillvalue*. Iteration continues until the longest iterable is exhausted. Roughly equivalent to:

```
def zip_longest(*args, fillvalue=None):
    # zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
    iterators = [iter(it) for it in args]
    num_active = len(iterators)
    if not num_active:
        return
    while True:
        values = []
        for i, it in enumerate(iterators):
            try:
                value = next(it)
```

(continua na próxima página)

(continuação da página anterior)

```
            except StopIteration:
                num_active -= 1
                if not num_active:
                    return
                iterators[i] = repeat(fillvalue)
                value = fillvalue
            values.append(value)
        yield tuple(values)
```

Se um dos iteráveis é potencialmente infinito, então a função *zip_longest()* deve ser embrulhada por algo que limite o número de chamadas (por exemplo *islice()* ou *takewhile()*). Se não especificado, *fillvalue* tem o valor padrão `None`.

10.1.2 Receitas com itertools

Esta seção mostra receitas para criação de um ferramental ampliado usando as ferramentas existentes de itertools como elementos construtivos.

The extended tools offer the same high performance as the underlying toolset. The superior memory performance is kept by processing elements one at a time rather than bringing the whole iterable into memory all at once. Code volume is kept small by linking the tools together in a functional style which helps eliminate temporary variables. High speed is retained by preferring "vectorized" building blocks over the use of for-loops and *generator*s which incur interpreter overhead.

```
def take(n, iterable):
    "Return first n items of the iterable as a list"
    return list(islice(iterable, n))

def prepend(value, iterator):
    "Prepend a single value in front of an iterator"
    # prepend(1, [2, 3, 4]) -> 1 2 3 4
    return chain([value], iterator)

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return map(function, count(start))

def tail(n, iterable):
    "Return an iterator over the last n items"
    # tail(3, 'ABCDEFG') --> E F G
    return iter(collections.deque(iterable, maxlen=n))

def consume(iterator, n=None):
    "Advance the iterator n-steps ahead. If n is None, consume entirely."
    # Use functions that consume iterators at C speed.
    if n is None:
        # feed the entire iterator into a zero-length deque
        collections.deque(iterator, maxlen=0)
    else:
        # advance to the empty slice starting at position n
        next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
    "Returns the nth item or a default value"
    return next(islice(iterable, n, None), default)
```

(continua na próxima página)

(continuação da página anterior)

```
def all_equal(iterable):
    "Returns True if all the elements are equal to each other"
    g = groupby(iterable)
    return next(g, True) and not next(g, False)

def quantify(iterable, pred=bool):
    "Count how many times the predicate is true"
    return sum(map(pred, iterable))

def padnone(iterable):
    """Returns the sequence elements and then returns None indefinitely.

    Useful for emulating the behavior of the built-in map() function.
    """
    return chain(iterable, repeat(None))

def ncycles(iterable, n):
    "Returns the sequence elements n times"
    return chain.from_iterable(repeat(tuple(iterable), n))

def dotproduct(vec1, vec2):
    return sum(map(operator.mul, vec1, vec2))

def flatten(listOfLists):
    "Flatten one level of nesting"
    return chain.from_iterable(listOfLists)

def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments.

    Example:  repeatfunc(random.random)
    """
    if times is None:
        return starmap(func, repeat(args))
    return starmap(func, repeat(args, times))

def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b, None)
    return zip(a, b)

def grouper(iterable, n, fillvalue=None):
    "Collect data into fixed-length chunks or blocks"
    # grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx"
    args = [iter(iterable)] * n
    return zip_longest(*args, fillvalue=fillvalue)

def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    # Recipe credited to George Sakkis
    num_active = len(iterables)
    nexts = cycle(iter(it).__next__ for it in iterables)
    while num_active:
        try:
            for next in nexts:
                yield next()
```

(continua na próxima página)

(continuação da página anterior)

```
        except StopIteration:
            # Remove the iterator we just exhausted from the cycle.
            num_active -= 1
            nexts = cycle(islice(nexts, num_active))

def partition(pred, iterable):
    'Use a predicate to partition entries into false entries and true entries'
    # partition(is_odd, range(10)) --> 0 2 4 6 8   and  1 3 5 7 9
    t1, t2 = tee(iterable)
    return filterfalse(pred, t1), filter(pred, t2)

def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def unique_everseen(iterable, key=None):
    "List unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') --> A B C D
    # unique_everseen('ABBCcAD', str.lower) --> A B C D
    seen = set()
    seen_add = seen.add
    if key is None:
        for element in filterfalse(seen.__contains__, iterable):
            seen_add(element)
            yield element
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen_add(k)
                yield element

def unique_justseen(iterable, key=None):
    "List unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') --> A B C D A B
    # unique_justseen('ABBCcAD', str.lower) --> A B C A D
    return map(next, map(itemgetter(1), groupby(iterable, key)))

def iter_except(func, exception, first=None):
    """ Call a function repeatedly until an exception is raised.

    Converts a call-until-exception interface to an iterator interface.
    Like builtins.iter(func, sentinel) but uses an exception instead
    of a sentinel to end the loop.

    Examples:
        iter_except(functools.partial(heappop, h), IndexError)    # priority queue␣
↪iterator
        iter_except(d.popitem, KeyError)                          # non-blocking dict␣
↪iterator
        iter_except(d.popleft, IndexError)                        # non-blocking deque␣
↪iterator
        iter_except(q.get_nowait, Queue.Empty)                    # loop over a␣
↪producer Queue
        iter_except(s.pop, KeyError)                              # non-blocking set␣
↪iterator
```

(continua na próxima página)

(continuação da página anterior)

```
    """
    try:
        if first is not None:
            yield first()            # For database APIs needing an initial cast to␣
↪db.first()
        while True:
            yield func()
    except exception:
        pass

def first_true(iterable, default=False, pred=None):
    """Returns the first true value in the iterable.

    If no true value is found, returns *default*

    If *pred* is not None, returns the first item
    for which pred(item) is true.

    """
    # first_true([a,b,c], x) --> a or b or c or x
    # first_true([a,b], x, f) --> a if f(a) else b if f(b) else x
    return next(filter(pred, iterable), default)

def random_product(*args, repeat=1):
    "Random selection from itertools.product(*args, **kwds)"
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(random.choice(pool) for pool in pools)

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    "Random selection from itertools.combinations_with_replacement(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.randrange(n) for i in range(r))
    return tuple(pool[i] for i in indices)

def nth_combination(iterable, r, index):
    'Equivalent to list(combinations(iterable, r))[index]'
    pool = tuple(iterable)
    n = len(pool)
    if r < 0 or r > n:
        raise ValueError
    c = 1
    k = min(r, n-r)
```

(continua na próxima página)

(continuação da página anterior)

```
    for i in range(1, k+1):
        c = c * (n - k + i) // i
    if index < 0:
        index += c
    if index < 0 or index >= c:
        raise IndexError
    result = []
    while r:
        c, n, r = c*r//n, n-1, r-1
        while index >= c:
            index -= c
            c, n = c*(n-r)//n, n-1
        result.append(pool[-1-n])
    return tuple(result)
```

Note, many of the above recipes can be optimized by replacing global lookups with local variables defined as default values. For example, the *dotproduct* recipe can be written as:

```
def dotproduct(vec1, vec2, sum=sum, map=map, mul=operator.mul):
    return sum(map(mul, vec1, vec2))
```

10.2 `functools` — Funções e operações de ordem superior em objetos chamáveis

** Código fonte: **: source:*Lib/functools.py*

O módulo: mod: *functools* é para funções de ordem superior: funções que atuam ou retornam outras funções. Em geral, qualquer objeto invocável pode ser tratado como uma função para os propósitos deste módulo.

O módulo: mod:*functools* define as seguintes funções:

`functools.`**`cmp_to_key`**(*func*)

Transforma uma função de comparação de estilo antigo para um: termo: *função de chave '. Usado com ferramentas que aceitam funções-chave (como: func: 'sorted*,: func:' min',: func: *max*,: func:' heapq.nlargest',: func: *heapq.nsmallest*,: Func: *itertools.groupby*). Esta função é usada principalmente como uma ferramenta de transição para programas que estão sendo convertidos a partir do Python 2, que suportou o uso de funções de comparação.

Uma função de comparação é qualquer chamada que aceita dois argumentos, os compara e retorna um número negativo por menos de zero, igual a igualdade ou um número positivo por maior que. Uma função de chave é escalável que aceita um argumento e retorna outro valor para ser usado como a chave de classificação.

Exemplo:

```
sorted(iterable, key=cmp_to_key(locale.strcoll))  # locale-aware sort order
```

Para selecionar exemplos e um breve tutorial de classificação, veja: ref: *sortinghowto*.

Novo na versão 3.2.

`@functools.`**`lru_cache`**(*maxsize=128*, *typed=False*)

Decorador para embrulhar uma função com memorizável é chamada e que economiza até as chamadas mais recentes * maxsize *. Pode economizar tempo quando uma função cara ou I/O é periodicamente chamada com os mesmos argumentos.

dicionárioUma vez que um dicionário é usado para armazenar resultados em cache, os argumentos posicionais e de palavras-chave para a função devem ser hashable.

Padrões de argumento distintos podem ser considerados chamadas distintas com entradas de cache separadas. Por exemplo, *f(a=1, b=2)* e *f(b=2, a=1)* diferem em sua ordem de argumento nomeado e podem ter duas entradas de cache separadas.

Se * maxsize * estiver configurado para " None", o recurso LRU está desabilitado e o cache pode crescer sem ligação. O recurso LRU funciona melhor quando * maxsize * é um poder de dois.

Se *tipo* for definido como verdadeiro, os argumentos de função de diferentes tipos serão armazenados em cache separadamente. Por exemplo, " f (3) " e " f (3.0) " serão tratados como chamadas distintas com resultados distintos.

Para ajudar a medir a eficácia do cache e ajustar o parâmetro *maxsize*, a função envolvida é instrumentada com uma função: func: *cache_info* que retorna um: termo:' nomeado tuple' mostrando * hits *, * misses *, * Maxsize * e * currsize *. Em um ambiente multi-threaded, os hits e erros são aproximados.

O decorador também fornece uma função: func: *cache_clear* para limpar ou invalidar o cache.

A função subjacente original é acessível através do atributo: attr: *__wrapped__*. Isso é útil para introspecção, para ignorar o cache, ou para reinstalar a função com um cache diferente.

Um cache *LRU (least recently used - em português - menos usado recentemente) <https://en.wikipedia.org/wiki/Cache_algorithms#Examples>* _ funciona melhor quando as chamadas mais recentes são os melhores preditores de chamadas futuras (por exemplo, os artigos mais populares em um servidor de notícias tendem a mudar a cada dia). O limite de tamanho do cache garante que o cache não cresça sem está ligado a processos de longa duração, como servidores web.

Em geral, o cache LRU deve ser usado somente quando você deseja reutilizar valores calculados anteriormente. Da mesma forma, não faz sentido armazenar em cache funções com efeitos colaterais, funções que precisam criar objetos mutáveis distintos em cada chamada ou funções impuras, como time() ou random().

Exemplo de um cache LRU para conteúdo web estático

```
@lru_cache(maxsize=32)
def get_pep(num):
    'Retrieve text of a Python Enhancement Proposal'
    resource = 'http://www.python.org/dev/peps/pep-%04d/' % num
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...     pep = get_pep(n)
...     print(n, len(pep))

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

Exemplo de computação eficiente dos números Fibonacci <https://en.wikipedia.org/wiki/Fibonacci_number> _ *usando um cache para implementar uma" programação dinâmica "<https://en.wikipedia.org/wiki/Dynamic_programming>* _ técnica:

```
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

(continua na próxima página)

(continuação da página anterior)

```
>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```

Novo na versão 3.2.

Alterado na versão 3.3: Adicionado a opção * digitado *.

`@functools.total_ordering`

Given a class defining one or more rich comparison ordering methods, this class decorator supplies the rest. This simplifies the effort involved in specifying all of the possible rich comparison operations:

The class must define one of `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`. In addition, the class should supply an `__eq__()` method.

Por exemplo:

```
@total_ordering
class Student:
    def _is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

Nota: While this decorator makes it easy to create well behaved totally ordered types, it *does* come at the cost of slower execution and more complex stack traces for the derived comparison methods. If performance benchmarking indicates this is a bottleneck for a given application, implementing all six rich comparison methods instead is likely to provide an easy speed boost.

Novo na versão 3.2.

Alterado na versão 3.4: Returning NotImplemented from the underlying comparison function for unrecognised types is now supported.

`functools.partial`(*func*, **args*, ***keywords*)

Return a new *partial object* which when called will behave like *func* called with the positional arguments *args* and keyword arguments *keywords*. If more arguments are supplied to the call, they are appended to *args*. If additional keyword arguments are supplied, they extend and override *keywords*. Roughly equivalent to:

```
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*args, *fargs, **newkeywords)
```

(continua na próxima página)

(continuação da página anterior)

```
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

The *partial()* is used for partial function application which "freezes" some portion of a function's arguments and/or keywords resulting in a new object with a simplified signature. For example, *partial()* can be used to create a callable that behaves like the *int()* function where the *base* argument defaults to two:

```
>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

class functools.**partialmethod**(*func*, **args*, ***keywords*)

Return a new *partialmethod* descriptor which behaves like *partial* except that it is designed to be used as a method definition rather than being directly callable.

func must be a *descriptor* or a callable (objects which are both, like normal functions, are handled as descriptors).

When *func* is a descriptor (such as a normal Python function, *classmethod()*, *staticmethod()*, abstractmethod() or another instance of *partialmethod*), calls to __get__ are delegated to the underlying descriptor, and an appropriate *partial object* returned as the result.

When *func* is a non-descriptor callable, an appropriate bound method is created dynamically. This behaves like a normal Python function when used as a method: the *self* argument will be inserted as the first positional argument, even before the *args* and *keywords* supplied to the *partialmethod* constructor.

Exemplo:

```
>>> class Cell(object):
...     def __init__(self):
...         self._alive = False
...     @property
...     def alive(self):
...         return self._alive
...     def set_state(self, state):
...         self._alive = bool(state)
...     set_alive = partialmethod(set_state, True)
...     set_dead = partialmethod(set_state, False)
...
>>> c = Cell()
>>> c.alive
False
>>> c.set_alive()
>>> c.alive
True
```

Novo na versão 3.4.

functools.**reduce**(*function*, *iterable*[, *initializer*])

Apply *function* of two arguments cumulatively to the items of *sequence*, from left to right, so as to reduce the sequence to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates (((((1+2)+3)+4)+5). The left argument, *x*, is the accumulated value and the right argument, *y*, is the update value from the *sequence*. If the optional *initializer* is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty. If *initializer* is not given and *sequence* contains only one item, the first item is returned.

Aproximadamente equivalente a:

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

`@functools.`**`singledispatch`**

Transform a function into a *single-dispatch generic function*.

To define a generic function, decorate it with the `@singledispatch` decorator. Note that the dispatch happens on the type of the first argument, create your function accordingly:

```
>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg, verbose=False):
...     if verbose:
...         print("Let me just say,", end=" ")
...     print(arg)
```

To add overloaded implementations to the function, use the `register()` attribute of the generic function. It is a decorator. For functions annotated with types, the decorator will infer the type of the first argument automatically:

```
>>> @fun.register
... def _(arg: int, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> @fun.register
... def _(arg: list, verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

For code which doesn't use type annotations, the appropriate type argument can be passed explicitly to the decorator itself:

```
>>> @fun.register(complex)
... def _(arg, verbose=False):
...     if verbose:
...         print("Better than complicated.", end=" ")
...     print(arg.real, arg.imag)
...
```

To enable registering lambdas and pre-existing functions, the `register()` attribute can be used in a functional form:

```
>>> def nothing(arg, verbose=False):
...     print("Nothing.")
...
>>> fun.register(type(None), nothing)
```

The `register()` attribute returns the undecorated function which enables decorator stacking, pickling, as well as creating unit tests for each variant independently:

```
>>> @fun.register(float)
... @fun.register(Decimal)
... def fun_num(arg, verbose=False):
...     if verbose:
...         print("Half of your number:", end=" ")
...     print(arg / 2)
...
>>> fun_num is fun
False
```

When called, the generic function dispatches on the type of the first argument:

```
>>> fun("Hello, world.")
Hello, world.
>>> fun("test.", verbose=True)
Let me just say, test.
>>> fun(42, verbose=True)
Strength in numbers, eh? 42
>>> fun(['spam', 'spam', 'eggs', 'spam'], verbose=True)
Enumerate this:
0 spam
1 spam
2 eggs
3 spam
>>> fun(None)
Nothing.
>>> fun(1.23)
0.615
```

Where there is no registered implementation for a specific type, its method resolution order is used to find a more generic implementation. The original function decorated with `@singledispatch` is registered for the base `object` type, which means it is used if no better implementation is found.

To check which implementation will the generic function choose for a given type, use the `dispatch()` attribute:

```
>>> fun.dispatch(float)
<function fun_num at 0x1035a2840>
>>> fun.dispatch(dict)    # note: default implementation
<function fun at 0x103fe0000>
```

To access all registered implementations, use the read-only `registry` attribute:

```
>>> fun.registry.keys()
dict_keys([<class 'NoneType'>, <class 'int'>, <class 'object'>,
           <class 'decimal.Decimal'>, <class 'list'>,
           <class 'float'>])
>>> fun.registry[float]
<function fun_num at 0x1035a2840>
>>> fun.registry[object]
<function fun at 0x103fe0000>
```

Novo na versão 3.4.

Alterado na versão 3.7: The `register()` attribute supports using type annotations.

`functools.`**`update_wrapper`**(*wrapper*, *wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

Update a *wrapper* function to look like the *wrapped* function. The optional arguments are tuples to specify which attributes of the original function are assigned directly to the matching attributes on the wrapper function and which attributes of the wrapper function are updated with the corresponding attributes from the original function. The default values for these arguments are the module level constants `WRAPPER_ASSIGNMENTS` (which assigns to the wrapper function's `__module__`, `__name__`, `__qualname__`, `__annotations__` and `__doc__`, the documentation string) and `WRAPPER_UPDATES` (which updates the wrapper function's `__dict__`, i.e. the instance dictionary).

To allow access to the original function for introspection and other purposes (e.g. bypassing a caching decorator such as *lru_cache()*), this function automatically adds a `__wrapped__` attribute to the wrapper that refers to the function being wrapped.

The main intended use for this function is in *decorator* functions which wrap the decorated function and return the wrapper. If the wrapper function is not updated, the metadata of the returned function will reflect the wrapper definition rather than the original function definition, which is typically less than helpful.

update_wrapper() may be used with callables other than functions. Any attributes named in *assigned* or *updated* that are missing from the object being wrapped are ignored (i.e. this function will not attempt to set them on the wrapper function). *AttributeError* is still raised if the wrapper function itself is missing any attributes named in *updated*.

Novo na versão 3.2: Automatic addition of the `__wrapped__` attribute.

Novo na versão 3.2: Copying of the `__annotations__` attribute by default.

Alterado na versão 3.2: Missing attributes no longer trigger an *AttributeError*.

Alterado na versão 3.4: The `__wrapped__` attribute now always refers to the wrapped function, even if that function defined a `__wrapped__` attribute. (see bpo-17482)

`@functools.`**`wraps`**(*wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

This is a convenience function for invoking *update_wrapper()* as a function decorator when defining a wrapper function. It is equivalent to `partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)`. For example:

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print('Calling decorated function')
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print('Called example function')
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

Without the use of this decorator factory, the name of the example function would have been `'wrapper'`, and the docstring of the original `example()` would have been lost.

10.2.1 Objetos `partial`

partial objects are callable objects created by *partial()*. They have three read-only attributes:

partial.**func**
: A callable object or function. Calls to the *partial* object will be forwarded to *func* with new arguments and keywords.

partial.**args**
: The leftmost positional arguments that will be prepended to the positional arguments provided to a *partial* object call.

partial.**keywords**
: The keyword arguments that will be supplied when the *partial* object is called.

partial objects are like `function` objects in that they are callable, weak referencable, and can have attributes. There are some important differences. For instance, the *__name__* and `__doc__` attributes are not created automatically. Also, *partial* objects defined in classes behave like static methods and do not transform into bound methods during instance attribute look-up.

10.3 `operator` — Operadores padrão como funções

Código-fonte: Lib/operator.py

O módulo *operator* exporta um conjunto de funções eficientes correspondentes aos operadores intrínsecos do Python. Por exemplo, `operator.add(x,y)` é equivalente à expressão `x+y`. Muitos nomes de função são aqueles usados para métodos especiais, sem os sublinhados duplos. Para compatibilidade com versões anteriores, muitos deles têm uma variante com os sublinhados duplos mantidos. As variantes sem os sublinhados duplos são preferenciais para maior clareza.

As funções se enquadram em categorias que realizam comparações de objetos, operações lógicas, operações matemáticas e operações de sequência.

As funções de comparação de objetos são úteis para todos os objetos e são nomeadas após os operadores de comparação que os mesmos suportam:

operator.**lt**(*a*, *b*)
operator.**le**(*a*, *b*)
operator.**eq**(*a*, *b*)
operator.**ne**(*a*, *b*)
operator.**ge**(*a*, *b*)
operator.**gt**(*a*, *b*)
operator.**__lt__**(*a*, *b*)
operator.**__le__**(*a*, *b*)
operator.**__eq__**(*a*, *b*)
operator.**__ne__**(*a*, *b*)
operator.**__ge__**(*a*, *b*)
operator.**__gt__**(*a*, *b*)
: Executa “comparações ricas” entre *a* e *b*. Especialmente, `lt(a, b)` é equivalente a `a < b`, `le(a, b)` é equivalente a `a <= b`, `eq(a, b)` é equivalente a `a == b`, `ne(a, b)` é equivalente a `a != b`, `gt(a, b)` é equivalente a `a > b` e `ge(a, b)` é equivalente a `a >= b`. Observe que essas funções podem retornar qualquer valor, que pode ou não ser interpretável como um valor booleano. Consulte comparisons para obter mais informações sobre comparações ricas. Como as funções de comparação de objetos são úteis para todos os objetos e são nomeados pelos sistemas de comparação.

As operações lógicas também são geralmente aplicáveis a todos os objetos e suportam testes de verdade, testes de identidade e operações booleanas:

operator.**not_**(*obj*)
operator.**__not__**(*obj*)
Retorna o resultado de `not` *obj*. (Veja que não existe nenhum método *__not__()* para a instância do objetos; apenas o núcleo do interpretador definirá esta operação. O resultado será afetado pelo `__bool__()` e `__len__()` methods.)

operator.**truth**(*obj*)
Retorna *True* se o *obj* for True, e *False* caso contrário. Isso é equivalente a utilizar a construção *bool*.

operator.**is_**(*a*, *b*)
Retorna `a is b`. Testa a identidade do objeto.

operator.**is_not**(*a*, *b*)
Retorna `a is not b`. Testa a identidade do objeto.

As operações matemáticas bit a bit são as mais numerosas:

operator.**abs**(*obj*)
operator.**__abs__**(*obj*)
Retorna o valor absoluto de *obj*.

operator.**add**(*a*, *b*)
operator.**__add__**(*a*, *b*)
Retorna `a + b`, onde *a* e *b* são números

operator.**and_**(*a*, *b*)
operator.**__and__**(*a*, *b*)
Retornar bit a bit de *a* e *b*.

operator.**floordiv**(*a*, *b*)
operator.**__floordiv__**(*a*, *b*)
Retorna `a // b`.

operator.**index**(*a*)
operator.**__index__**(*a*)
Retorna *a* convertendo para um inteiro. Equivalente a"a.__index__()".

operator.**inv**(*obj*)
operator.**invert**(*obj*)
operator.**__inv__**(*obj*)
operator.**__invert__**(*obj*)
Retorna o inverso bit a bit do número *obj*. Isso equivale a `~obj`.

operator.**lshift**(*a*, *b*)
operator.**__lshift__**(*a*, *b*)
Retorna *a* deslocado para a esquerda por *b*.

operator.**mod**(*a*, *b*)
operator.**__mod__**(*a*, *b*)
Retorna `a % b`.

operator.**mul**(*a*, *b*)
operator.**__mul__**(*a*, *b*)
Retorna `a * b`, onde *a* e *b* são números

operator.**matmul**(*a*, *b*)
operator.**__matmul__**(*a*, *b*)
Retorna `a @ b`.

Novo na versão 3.5.

operator.**neg**(*obj*)
operator.**__neg__**(*obj*)
Retorna *obj* negado (`-obj`).

operator.**or_**(*a*, *b*)
operator.**__or__**(*a*, *b*)
Retorna bit a bit de *a* e *b*.

operator.**pos**(*obj*)
operator.**__pos__**(*obj*)
Retorna *obj* positivo (`+obj`).

operator.**pow**(*a*, *b*)
operator.**__pow__**(*a*, *b*)
Retorna `a ** b`, onde *a* e *b* são números.

operator.**rshift**(*a*, *b*)
operator.**__rshift__**(*a*, *b*)
Retorna *a* deslocado para a direita por *b*.

operator.**sub**(*a*, *b*)
operator.**__sub__**(*a*, *b*)
Retorna `a - b`.

operator.**truediv**(*a*, *b*)
operator.**__truediv__**(*a*, *b*)
Retorna `a / b` onde 2/3 é .66 em vez de 0. Isso também é conhecido como divisão “verdadeira”.

operator.**xor**(*a*, *b*)
operator.**__xor__**(*a*, *b*)
Retornar o bitwise exclusivo de *a* e *b*.

Operações que funcionam com sequências (algumas delas com mapas também) incluem:

operator.**concat**(*a*, *b*)
operator.**__concat__**(*a*, *b*)
Retorna `a + b` para as sequências *a* e *b*.

operator.**contains**(*a*, *b*)
operator.**__contains__**(*a*, *b*)
Retorna o resultado do teste `b in a`. Observe os operandos invertidos.

operator.**countOf**(*a*, *b*)
Retorna o número de ocorrências de *b* em *a*.

operator.**delitem**(*a*, *b*)
operator.**__delitem__**(*a*, *b*)
Remove o valor de *a* no índice *b*.

operator.**getitem**(*a*, *b*)
operator.**__getitem__**(*a*, *b*)
Retorna o valor de *a* no índice *b*.

operator.**indexOf**(*a*, *b*)
Retorna o índice da primeira ocorrência de *b* em *a*.

operator.**setitem**(*a*, *b*, *c*)
operator.**__setitem__**(*a*, *b*, *c*)
Define o valor de *a* no índice *b* para *c*.

operator.**length_hint**(*obj*, *default=0*)
Retorna um comprimento estimado para o objeto *o*. Primeiro tente retornar o seu comprimento real, em seguida, uma estimativa utilizando `object.__length_hint__()`, e finalmente retorna o valor default.

Novo na versão 3.4.

O módulo *operator* also defines tools for generalized attribute and item lookups. Estes são úteis para fazer extração de campo rapidamente como argumentos para a função *map()*, *sorted()*, *itertools.groupby()*, ou outra função que espera um argumento de função.

operator.**attrgetter**(*attr*)
operator.**attrgetter**(**attrs*)
Retorna um objeto invocável que pode buscar o *attr* do seu operando. Caso seja solicitado mais de um atributo, retorna uma tupla de atributos. Os nomes dos atributos também podem conter pontos. Por exemplo:

- Depois de `f = attrgetter('name')`, a chamada a `f(b)` retorna `b.name`.
- Depois de `f = attrgetter('name', 'date')`, a chamada a `f(b)` retorna"(b.name, b.date)".
- Depois de `f = attrgetter('name.first', 'name.last')`, a chamada a `f(b)` retorna `(b.name.first, b.name.last)`.

Equivalente a:

```
def attrgetter(*items):
    if any(not isinstance(item, str) for item in items):
        raise TypeError('attribute name must be a string')
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g

def resolve_attr(obj, attr):
    for name in attr.split("."):
        obj = getattr(obj, name)
    return obj
```

operator.**itemgetter**(*item*)
operator.**itemgetter**(**items*)
Retornar um objeto callable que busca *item* de seu operando usando o operando do método *__getitem__()*. Se múltiplo itens são especificados, retorna uma tupla de valores da pesquisa. Por exemplo:

- Depois de `f = itemgetter(2)`, a chamada a `f(r)` retorna `r[2]`.
- Depois de `g = itemgetter(2, 5, 3)`, a chamada a `g(r)` retorna `(r[2], r[5], r[3])`.

Equivalente a:

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

Os itens podem ser qualquer tipo aceito pelo método *__getitem__()*. Os dicionários aceitam qualquer valor hashable. Listas, tuplas e strings aceitam um índice ou uma fatia:

```
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1,3,5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2,None))('ABCDEFG')
'CDEFG'
```

```
>>> soldier = dict(rank='captain', name='dotterbart')
>>> itemgetter('rank')(soldier)
'captain'
```

Exemplo de uso *itemgetter()* para recuperar campos específicos de um registro de tupla:

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> list(map(getcount, inventory))
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

operator.**methodcaller**(*name*[, *args...*])

Retornar um objeto callable que invoca o método *name* em seu operando. Se argumentos adicionais e/ou argumentos de keyword forem fornecidos, os mesmos serão passados para o método. Por exemplo:

- Depois de `f = methodcaller('name')`, a chamada a `f(b)` retorna `b.name()`.
- Depois de `f = methodcaller('name', 'foo', bar=1)`, a chamada `f(b)` retorna `b.name('foo', bar=1)`.

Equivalente a:

```
def methodcaller(name, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

10.3.1 Mapeando os Operadores para suas Respectivas Funções

Esta tabela mostra como as operações abstratos correspondem aos símbolos do operador na sintaxe Python e as funções no módulo *operator*.

Operação	Sintaxe	Função
Adiçao	`a + b`	`add(a, b)`
Concatenação	`seq1 + seq2`	`concat(seq1, seq2)`
Teste de Contenção	`obj in seq`	`contains(seq, obj)`
Divisão	`a / b`	`truediv(a, b)`
Divisão	`a // b`	`floordiv(a, b)`
Bitwise And	`a & b`	`and_(a, b)`
Bitwise Exclusivo Or	`a ^ b`	`xor(a, b)`
Bitwise Inversão	`~ a`	`invert(a)`
Bitwise Or	`a | b`	`or_(a, b)`

Continuação na próxima página

Tabela 1 – continuação da página anterior

Operação	Sintaxe	Função
Exponenciação	`a ** b`	`pow(a, b)`
Identidade	`a is b`	`is_(a, b)`
Identidade	`a is not b`	`is_not(a, b)`
Atribuição Indexada	`obj[k] = v`	`setitem(obj, k, v)`
Eliminação Indexada	`del obj[k]`	`delitem(obj, k)`
Indexação	`obj[k]`	`getitem(obj, k)`
Left Shift	`a << b`	`lshift(a, b)`
Módulo	`a % b`	`mod(a, b)`
Multiplicação	`a * b`	`mul(a, b)`
Multiplicação de Arrays	`a @ b`	`matmul(a, b)`
Negação (Aritmética)	`- a`	`neg(a)`
Negaçaõ (Logical)	`not a`	`not_(a)`
Positivo	`+ a`	`pos(a)`
Right Shift	`a >> b`	`rshift(a, b)`
Atribuição de Fatiamento	`seq[i:j] = values`	`setitem(seq, slice(i, j), values)`
Remoção de Fatiamento	`del seq[i:j]`	`delitem(seq, slice(i, j))`
Fatiamento	`seq[i:j]`	`getitem(seq, slice(i, j))`
Formatação de Strings	`s % obj`	`mod(s, obj)`
Subtração	`a - b`	`sub(a, b)`
Teste Verdadeiro	`obj`	`truth(obj)`
Ordenação	`a < b`	`lt(a, b)`
Ordenação	`a <= b`	`le(a, b)`
Igualdade	`a == b`	`eq(a, b)`
Diferença	`a != b`	`ne(a, b)`
Ordenação	`a >= b`	`ge(a, b)`
Ordenação	`a > b`	`gt(a, b)`

10.3.2 Operadores in-place

Muitas operações possuem uma versão "in-place". Listadas abaixo, as funções fornecem um acesso mais primitivo as operadores locais do que a sintaxe usual; por exemplo, o termo *statement* `x += y` é equivalente a `x = operator.iadd(x, y)`. Outra maneira de colocá-lo é dizendo que `z = operator.iadd(x, y)` é equivalente à instrução composta `z = x; z += y`.

Nesses exemplos, note que, quando um método in-place é invocado, a computação e a atribuição são realizadas em duas etapas separadas. As funções in-lace listadas abaixo apenas fazem o primeiro passo, invocando o método in-place. O segundo passo, a atribuição, não é tratado.

Para os casos imutáveis, como as Strings, números e tuplas, o valor atualizado será calculado, mas não será atribuído de volta à variável de entrada:

```
>>> a = 'hello'
>>> iadd(a, ' world')
'hello world'
>>> a
'hello'
```

Para alvos mutáveis tal como listas e dicionários, o método in-place vai realizar a atualização, então nenhuma atribuição subsequente é necessária:

```
>>> s = ['h', 'e', 'l', 'l', 'o']
>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> s
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

operator.**iadd**(*a*, *b*)
operator.**__iadd__**(*a*, *b*)
a = iadd(a, b) is equivalent to a += b.

operator.**iand**(*a*, *b*)
operator.**__iand__**(*a*, *b*)
a = iand(a, b) is equivalent to a &= b.

operator.**iconcat**(*a*, *b*)
operator.**__iconcat__**(*a*, *b*)
a = iconcat(a, b) é equivalente a a += b onde *a* e *b* são sequências.

operator.**ifloordiv**(*a*, *b*)
operator.**__ifloordiv__**(*a*, *b*)
a = ifloordiv(a, b) é equivalente a a //= b.

operator.**ilshift**(*a*, *b*)
operator.**__ilshift__**(*a*, *b*)
a = ilshift(a, b) é equivalente a a <<= b.

operator.**imod**(*a*, *b*)
operator.**__imod__**(*a*, *b*)
a = imod(a, b) é equivalente a a %= b.

operator.**imul**(*a*, *b*)
operator.**__imul__**(*a*, *b*)
a = imul(a, b) é equivalente a a *= b.

operator.**imatmul**(*a*, *b*)
operator.**__imatmul__**(*a*, *b*)
a = imatmul(a, b) é equivalente a a @= b.

Novo na versão 3.5.

operator.**ior**(*a*, *b*)
operator.**__ior__**(*a*, *b*)
a = ior(a, b) é equivalente a a |= b.

operator.**ipow**(*a*, *b*)
operator.**__ipow__**(*a*, *b*)
a = ipow(a, b) é equivalente a a **= b.

operator.**irshift**(*a*, *b*)
operator.**__irshift__**(*a*, *b*)
a = irshift(a, b) é equivalente a a >>= b.

operator.**isub**(*a*, *b*)
operator.**__isub__**(*a*, *b*)
a = isub(a, b) é equivalente a a -= b.

operator.**itruediv**(*a*, *b*)
operator.**__itruediv__**(*a*, *b*)
a = itruediv(a, b) é equivalente a a /= b.

operator.**ixor**(*a*, *b*)

operator.**__ixor__**(*a*, *b*)
    `a = ixor(a, b)` é equivalente a `a ^= b`.

CAPÍTULO 11

Arquivo e Acesso aos Diretórios

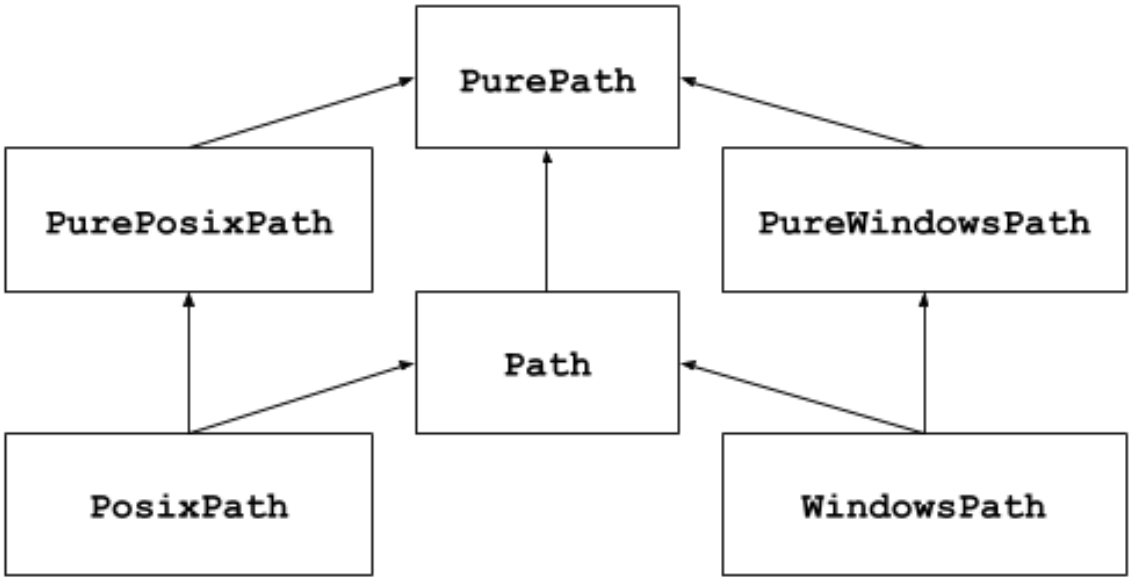
Os módulos descritos neste capítulo dizem respeitos aos arquivos e diretórios no disco. Por exemplo, existem módulos para ler as propriedades dos arquivos, manipular o caminhos de forma multiplataforma e para criar arquivos temporários. A lista completa de módulos neste capítulo é:

11.1 `pathlib` — Caminhos do Sistema de Arquivos Orientados a Objetos

Novo na versão 3.4.

Código Fonte: Lib/pathlib.py

Este módulo oferece classes que representam caminhos de sistema de arquivos com semântica apropriada para diferentes sistemas operacionais. As classes de caminho são divididas entre *caminhos puros*, que fornecem operações puramente computacionais sem E/S, e *caminhos concretos*, que herdam de caminhos puros, mas também fornecem operações de E/S.



Se você nunca usou este módulo antes ou apenas não tem certeza de qual classe é a certa para sua tarefa, provavelmente `Path` é o que você precisa. Ele instancia um *caminho concreto* para a plataforma em que o código está sendo executado.

Caminhos puros são úteis em alguns casos especiais. Por exemplo:

1. Se você deseja manipular os caminhos do Windows em uma máquina Unix (ou vice-versa). Você não pode instanciar uma `WindowsPath` quando executado no Unix, mas você pode instanciar `PureWindowsPath`.
2. Você quer ter certeza de que seu código apenas manipula caminhos, sem realmente acessar o sistema operacional. Nesse caso, instanciar uma das classes puras pode ser útil, pois elas simplesmente não têm nenhuma operação de acesso ao sistema operacional.

Ver também:

PEP 428: O módulo pathlib – caminhos de sistema de arquivos orientados a objetos.

Ver também:

Para manipulação de caminho de baixo nível em strings, você também pode usar o módulo `os.path`.

11.1.1 Utilização Básica

Importação da classe principal:

```
>>> from pathlib import Path
```

Listando os subdiretórios:

```
>>> p = Path('.')
>>> [x for x in p.iterdir() if x.is_dir()]
[PosixPath('.hg'), PosixPath('docs'), PosixPath('dist'),
 PosixPath('__pycache__'), PosixPath('build')]
```

Listando os arquivos fontes do Python e sua árvore de diretórios:

```
>>> list(p.glob('**/*.py'))
[PosixPath('test_pathlib.py'), PosixPath('setup.py'),
 PosixPath('pathlib.py'), PosixPath('docs/conf.py'),
 PosixPath('build/lib/pathlib.py')]
```

Navegando dentro da árvore de diretórios:

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')
```

Consultando as propriedades do path:

```
>>> q.exists()
True
>>> q.is_dir()
False
```

Abrindo um arquivo:

```
>>> with q.open() as f: f.readline()
...
'#!/bin/bash\n'
```

11.1.2 Caminhos puros

Objetos de caminho puro fornecem operações de manipulação de caminho que, na verdade, não acessam um sistema de arquivos. Existem três maneiras de acessar essas classes, que também chamamos de *sabores*:

class pathlib.**PurePath**(**pathsegments*)

Uma classe genérica que representa o tipo de caminho do sistema (instanciando-a cria uma *PurePosixPath* ou uma *PureWindowsPath*):

```
>>> PurePath('setup.py')      # Running on a Unix machine
PurePosixPath('setup.py')
```

Cada elemento de *pathsegments* pode ser uma string representando um segmento de caminho, um objeto que implementa a interface *os.PathLike* que retorna uma string ou outro objeto caminho:

```
>>> PurePath('foo', 'some/path', 'bar')
PurePosixPath('foo/some/path/bar')
>>> PurePath(Path('foo'), Path('bar'))
PurePosixPath('foo/bar')
```

Quando *pathsegments* está vazio, o diretório atual é presumido:

```
>>> PurePath()
PurePosixPath('.')
```

Quando vários caminhos absolutos são fornecidos, o último é tomado como uma âncora (imitando o comportamento de *os.path.join()*):

```
>>> PurePath('/etc', '/usr', 'lib64')
PurePosixPath('/usr/lib64')
>>> PureWindowsPath('c:/Windows', 'd:bar')
PureWindowsPath('d:bar')
```

No entanto, em um caminho do Windows, a alteração da raiz local não descarta a configuração da unidade anterior:

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

Barras espúrias e pontos únicos são recolhidos, mas os pontos duplos (`'..'`) não, pois isso mudaria o significado de um caminho em face de links simbólicos:

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/./bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/../bar')
PurePosixPath('foo/../bar')
```

(uma abordagem ingênua seria criar `PurePosixPath('foo/../bar')` equivalente a `PurePosixPath('bar')`, o que é errado se `foo` for um link simbólico para outro diretório)

Objetos caminho puro implementam a interface *os.PathLike*, permitindo que sejam usados em qualquer lugar em que a interface seja aceita.

Alterado na versão 3.6: Adicionado suporte para a interface *os.PathLike*.

class pathlib.**PurePosixPath**(**pathsegments*)

Uma subclasse de *PurePath*, este tipo de caminho representa caminhos de sistema de arquivos não Windows:

```
>>> PurePosixPath('/etc')
PurePosixPath('/etc')
```

pathsegments é especificado de forma similar para *PurePath*.

class pathlib.**PureWindowsPath**(**pathsegments*)

Uma subclasse de *PurePath*, este tipo de caminho representa os caminhos do sistema de arquivos do Windows:

```
>>> PureWindowsPath('c:/Program Files/')
PureWindowsPath('c:/Program Files')
```

pathsegments é especificado de forma similar para *PurePath*.

Independentemente do sistema em que você está usando, você pode instanciar todas essas classes, uma vez que elas não fornecem nenhuma operação que faça chamadas de sistema.

Propriedades Gerais

Os caminhos são imutáveis e hasheáveis. Os caminhos do mesmo sabor são comparáveis e ordenáveis. Essas propriedades respeitam a semântica de caixa alta e baixa do sabor:

```
>>> PurePosixPath('foo') == PurePosixPath('FOO')
False
>>> PureWindowsPath('foo') == PureWindowsPath('FOO')
True
>>> PureWindowsPath('FOO') in { PureWindowsPath('foo') }
```

(continua na próxima página)

(continuação da página anterior)

```
True
>>> PureWindowsPath('C:') < PureWindowsPath('d:')
True
```

Caminhos de um sabor diferente são comparados de forma desigual e não podem ser ordenados:

```
>>> PureWindowsPath('foo') == PurePosixPath('foo')
False
>>> PureWindowsPath('foo') < PurePosixPath('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'PureWindowsPath' and 'PurePosixPath
↪'
```

Operadores

O operador barra ajuda a criar caminhos filhos, de forma semelhante a *os.path.join()*:

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
```

Um objeto de caminho pode ser usado em qualquer lugar em que um objeto implementando *os.PathLike* seja aceito:

```
>>> import os
>>> p = PurePath('/etc')
>>> os.fspath(p)
'/etc'
```

A representação de string de um caminho é o próprio caminho do sistema de arquivos bruto (na forma nativa, por exemplo, com contrabarras no Windows), que você pode passar para qualquer função usando um caminho de arquivo como uma string:

```
>>> p = PurePath('/etc')
>>> str(p)
'/etc'
>>> p = PureWindowsPath('c:/Program Files')
>>> str(p)
'c:\\Program Files'
```

Da mesma forma, chamar *bytes* em um caminho fornece o caminho do sistema de arquivos bruto como um objeto bytes, codificado por *os.fsencode()*:

```
>>> bytes(p)
b'/etc'
```

Nota: A chamada de *bytes* só é recomendada no Unix. No Windows, a forma Unicode é a representação canônica dos caminhos do sistema de arquivos.

Acessando partes individuais

Para acessar as "partes" individuais (componentes) de um caminho, use a seguinte propriedade:

PurePath.**parts**
Uma tupla que dá acesso aos vários componentes do caminho:

```
>>> p = PurePath('/usr/bin/python3')
>>> p.parts
('/', 'usr', 'bin', 'python3')

>>> p = PureWindowsPath('c:/Program Files/PSF')
>>> p.parts
('c:\\', 'Program Files', 'PSF')
```

(observe como a unidade e a raiz local são reagrupadas em uma única parte)

Métodos e propriedades

Caminhos puros fornecem os seguintes métodos e propriedades:

PurePath.**drive**
Uma string que representa a letra ou nome da unidade, se houver:

```
>>> PureWindowsPath('c:/Program Files/').drive
'c:'
>>> PureWindowsPath('/Program Files/').drive
''
>>> PurePosixPath('/etc').drive
''
```

Os compartilhamentos UNC também são considerados unidades:

```
>>> PureWindowsPath('//host/share/foo.txt').drive
'\\\\host\\share'
```

PurePath.**root**
Uma string que representa a raiz (local ou global), se houver:

```
>>> PureWindowsPath('c:/Program Files/').root
'\\'
>>> PureWindowsPath('c:Program Files/').root
''
>>> PurePosixPath('/etc').root
'/'
```

Os compartilhamentos UNC sempre têm uma raiz:

```
>>> PureWindowsPath('//host/share').root
'\\'
```

PurePath.**anchor**
A concatenação da unidade e da raiz:

```
>>> PureWindowsPath('c:/Program Files/').anchor
'c:\\'
>>> PureWindowsPath('c:Program Files/').anchor
```

(continua na próxima página)

(continuação da página anterior)

```
'c:'
>>> PurePosixPath('/etc').anchor
'/'
>>> PureWindowsPath('//host/share').anchor
'\\\\host\\share\\'
```

PurePath.**parents**

Uma sequência imutável que fornece acesso aos ancestrais lógicos do caminho:

```
>>> p = PureWindowsPath('c:/foo/bar/setup.py')
>>> p.parents[0]
PureWindowsPath('c:/foo/bar')
>>> p.parents[1]
PureWindowsPath('c:/foo')
>>> p.parents[2]
PureWindowsPath('c:/')
```

PurePath.**parent**

O pai lógico do caminho:

```
>>> p = PurePosixPath('/a/b/c/d')
>>> p.parent
PurePosixPath('/a/b/c')
```

Você não pode passar por uma âncora ou caminho vazio:

```
>>> p = PurePosixPath('/')
>>> p.parent
PurePosixPath('/')
>>> p = PurePosixPath('.')
>>> p.parent
PurePosixPath('.')
```

Nota: Esta é uma operação puramente lexical, daí o seguinte comportamento:

```
>>> p = PurePosixPath('foo/..')
>>> p.parent
PurePosixPath('foo')
```

Se você quiser percorrer um caminho de sistema de arquivos arbitrário para cima, é recomendado primeiro chamar *`Path.resolve()`* para resolver links simbólicos e eliminar componentes "..".

PurePath.**name**

Uma string que representa o componente do caminho final, excluindo a unidade e a raiz, se houver:

```
>>> PurePosixPath('my/library/setup.py').name
'setup.py'
```

Nomes de unidades UNC não são considerados::

```
>>> PureWindowsPath('//some/share/setup.py').name
'setup.py'
>>> PureWindowsPath('//some/share').name
''
```

PurePath.**suffix**
A extensão do arquivo do componente final, se houver:

```
>>> PurePosixPath('my/library/setup.py').suffix
'.py'
>>> PurePosixPath('my/library.tar.gz').suffix
'.gz'
>>> PurePosixPath('my/library').suffix
''
```

PurePath.**suffixes**
Uma lista das extensões de arquivo do caminho:

```
>>> PurePosixPath('my/library.tar.gar').suffixes
['.tar', '.gar']
>>> PurePosixPath('my/library.tar.gz').suffixes
['.tar', '.gz']
>>> PurePosixPath('my/library').suffixes
[]
```

PurePath.**stem**
O componente final do caminho, sem seu sufixo:

```
>>> PurePosixPath('my/library.tar.gz').stem
'library.tar'
>>> PurePosixPath('my/library.tar').stem
'library'
>>> PurePosixPath('my/library').stem
'library'
```

PurePath.**as_posix**()
Retorna uma representação de string do caminho com barras (/):

```
>>> p = PureWindowsPath('c:\\windows')
>>> str(p)
'c:\\windows'
>>> p.as_posix()
'c:/windows'
```

PurePath.**as_uri**()
Representa o caminho como um URI de `file`. *ValueError* é levantada se o caminho não for absoluto.

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.as_uri()
'file:///etc/passwd'
>>> p = PureWindowsPath('c:/Windows')
>>> p.as_uri()
'file:///c:/Windows'
```

PurePath.**is_absolute**()
Retorna se o caminho é absoluto ou não. Um caminho é considerado absoluto se tiver uma raiz e (se o tipo permitir) uma unidade:

```
>>> PurePosixPath('/a/b').is_absolute()
True
>>> PurePosixPath('a/b').is_absolute()
False
```

(continua na próxima página)

(continuação da página anterior)

```
>>> PureWindowsPath('c:/a/b').is_absolute()
True
>>> PureWindowsPath('/a/b').is_absolute()
False
>>> PureWindowsPath('c:').is_absolute()
False
>>> PureWindowsPath('//some/share').is_absolute()
True
```

PurePath.**is_reserved**()

Com *PureWindowsPath*, retorna `True` se o caminho é considerado reservado no Windows, `False` caso contrário. Com *PurePosixPath*, `False` é sempre retornado.

```
>>> PureWindowsPath('nul').is_reserved()
True
>>> PurePosixPath('nul').is_reserved()
False
```

As chamadas do sistema de arquivos em caminhos reservados podem falhar misteriosamente ou ter efeitos indesejados.

PurePath.**joinpath**(**other*)

Chamar este método é equivalente a combinar o caminho com cada um dos outros argumentos, representados por *other*:

```
>>> PurePosixPath('/etc').joinpath('passwd')
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath(PurePosixPath('passwd'))
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath('init.d', 'apache2')
PurePosixPath('/etc/init.d/apache2')
>>> PureWindowsPath('c:').joinpath('/Program Files')
PureWindowsPath('c:/Program Files')
```

PurePath.**match**(*pattern*)

Compara esse caminho com o padrão de estilo glob fornecido. Retorna `True` se a correspondência for bem-sucedida, `False` caso contrário.

Se *pattern* for relativo, o caminho pode ser relativo ou absoluto, e a correspondência é feita da direita:

```
>>> PurePath('a/b.py').match('*.py')
True
>>> PurePath('/a/b/c.py').match('b/*.py')
True
>>> PurePath('/a/b/c.py').match('a/*.py')
False
```

Se *pattern* for absoluto, o caminho deve ser absoluto e todo o caminho deve corresponder a:

```
>>> PurePath('/a.py').match('/*.py')
True
>>> PurePath('a/b.py').match('/*.py')
False
```

Tal como acontece com outros métodos, a distinção entre maiúsculas e minúsculas segue os padrões da plataforma:

```
>>> PurePosixPath('b.py').match('*.PY')
False
>>> PureWindowsPath('b.py').match('*.PY')
True
```

PurePath.**relative_to**(**other*)

Calcula uma versão deste caminho em relação ao caminho representado por *other*. Se for impossível, ValueError é levantada:

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.relative_to('/')
PurePosixPath('etc/passwd')
>>> p.relative_to('/etc')
PurePosixPath('passwd')
>>> p.relative_to('/usr')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 694, in relative_to
    .format(str(self), str(formatted)))
ValueError: '/etc/passwd' does not start with '/usr'
```

PurePath.**with_name**(*name*)

Retorna um novo caminho com o *name* alterado. Se o caminho original não tiver um nome, ValueError é levantada:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_name('setup.py')
PureWindowsPath('c:/Downloads/setup.py')
>>> p = PureWindowsPath('c:/')
>>> p.with_name('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 751, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

PurePath.**with_suffix**(*suffix*)

Retorna um novo caminho com o *suffix* alterado. Se o caminho original não tiver um sufixo, o novo *suffixo* será anexado. Se o *suffix* for uma string vazia, o sufixo original será removido:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_suffix('.bz2')
PureWindowsPath('c:/Downloads/pathlib.tar.bz2')
>>> p = PureWindowsPath('README')
>>> p.with_suffix('.txt')
PureWindowsPath('README.txt')
>>> p = PureWindowsPath('README.txt')
>>> p.with_suffix('')
PureWindowsPath('README')
```

11.1.3 Caminhos Concretos

Caminhos concretos são subclasses das classes de caminho puro. Além das operações fornecidas por este último, eles também fornecem métodos para fazer chamadas de sistema em objetos de caminho. Existem três maneiras de instanciar caminhos concretos:

class pathlib.**Path**(**pathsegments*)

Uma subclasse de *PurePath*, esta classe representa caminhos concretos do tipo de caminho do sistema (instanciando-o cria uma *PosixPath* ou uma *WindowsPath*):

```
>>> Path('setup.py')
PosixPath('setup.py')
```

pathsegments é especificado de forma similar para *PurePath*.

class pathlib.**PosixPath**(**pathsegments*)

Uma subclasse de *Path* e *PurePosixPath*, esta classe representa caminhos concretos de sistemas de arquivos não Windows:

```
>>> PosixPath('/etc')
PosixPath('/etc')
```

pathsegments é especificado de forma similar para *PurePath*.

class pathlib.**WindowsPath**(**pathsegments*)

Uma subclasse de *Path* e *PureWindowsPath*, esta classe representa caminhos concretos de sistemas de arquivos do Windows:

```
>>> WindowsPath('c:/Program Files/')
WindowsPath('c:/Program Files')
```

pathsegments é especificado de forma similar para *PurePath*.

Você só pode instanciar o tipo de classe que corresponde ao seu sistema (permitir chamadas de sistema em tipos de caminho não compatíveis pode levar a bugs ou falhas em sua aplicação):

```
>>> import os
>>> os.name
'posix'
>>> Path('setup.py')
PosixPath('setup.py')
>>> PosixPath('setup.py')
PosixPath('setup.py')
>>> WindowsPath('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 798, in __new__
    % (cls.__name__,))
NotImplementedError: cannot instantiate 'WindowsPath' on your system
```

Métodos

Concrete paths provide the following methods in addition to pure paths methods. Many of these methods can raise an *OSError* if a system call fails (for example because the path doesn't exist):

classmethod Path.**cwd**()
Retorna um novo objeto de caminho que representa o diretório atual (conforme retornado por *os.getcwd()*):

```
>>> Path.cwd()
PosixPath('/home/antoine/pathlib')
```

classmethod Path.**home**()
Retorna um novo objeto de caminho representando o diretório inicial do usuário (conforme retornado por *os.path.expanduser()* com a construção ~):

```
>>> Path.home()
PosixPath('/home/antoine')
```

Novo na versão 3.5.

Path.**stat**()
Retorna um objeto *os.stat_result* contendo informações sobre este caminho, como *os.stat()*. O resultado é consultado em cada chamada para este método.

```
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

Path.**chmod**(*mode*)
Altera o modo de arquivo e as permissões, como *os.chmod()*:

```
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

Path.**exists**()
Se o caminho aponta para um arquivo ou diretório existente:

```
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

Nota: Se o caminho aponta para um link simbólico, *exists()* retorna se o link simbólico *aponta para* um arquivo ou diretório existente.

Path.**expanduser**()
Retorna um novo caminho com as construções expandidas `~` e `~user`, conforme retornado por *os.path.expanduser()*:

```
>>> p = PosixPath('~/films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

Novo na versão 3.5.

Path.**glob**(*pattern*)
Faz o glob do *pattern* relativo fornecido no diretório representado por este caminho, produzindo todos os arquivos correspondentes (de qualquer tipo):

```
>>> sorted(Path('.').glob('*.py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'), PosixPath('test_pathlib.py')]
>>> sorted(Path('.').glob('*/*.py'))
[PosixPath('docs/conf.py')]
```

O padrão "`**`" significa "este diretório e todos os subdiretórios, recursivamente". Em outras palavras, ele permite fazer glob recursivo:

```
>>> sorted(Path('.').glob('**/*.py'))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

Nota: Usar o padrão "`**`" em grandes árvores de diretório pode consumir uma quantidade excessiva de tempo.

Path.**group**()
Retorna o nome do grupo que possui o arquivo. *KeyError* é levantada se o gid do arquivo não for encontrado no banco de dados do sistema.

Path.**is_dir**()
Retorna `True` se o caminho apontar para um diretório (ou um link simbólico apontando para um diretório), `False` se apontar para outro tipo de arquivo.

`False` também é retornado se o caminho não existir ou se for um link simbólico quebrado; outros erros (como erros de permissão) são propagados.

Path.**is_file**()
Retorna `True` se o caminho apontar para um arquivo regular (ou um link simbólico apontando para um arquivo regular), `False` se apontar para outro tipo de arquivo.

`False` também é retornado se o caminho não existir ou se for um link simbólico quebrado; outros erros (como erros de permissão) são propagados.

Path.**is_mount**()
Retorna `True` se o caminho for um *ponto de montagem*: um ponto em um sistema de arquivos onde um sistema de arquivos diferente foi montado. No POSIX, a função verifica se o pai do *path*, `path/..`, está em um dispositivo diferente de *path*, ou se `path/..` e *path* apontam para o mesmo nó-i no mesmo dispositivo – isso deve detectar pontos de montagem para todas as variantes Unix e POSIX. Não implementado no Windows.

Novo na versão 3.7.

Path.**is_symlink**()
Retorna `True` se o caminho apontar para um link simbólico, `False` caso contrário.

`False` também é retornado se o caminho não existir; outros erros (como erros de permissão) são propagados.

Path.**is_socket**()
Retorna `True` se o caminho apontar para um soquete Unix (ou um link simbólico apontando para um soquete Unix), `False` se apontar para outro tipo de arquivo.

`False` também é retornado se o caminho não existir ou se for um link simbólico quebrado; outros erros (como erros de permissão) são propagados.

Path.**is_fifo**()
Retorna `True` se o caminho apontar para um FIFO (ou um link simbólico apontando para um FIFO), `False` se apontar para outro tipo de arquivo.

`False` também é retornado se o caminho não existir ou se for um link simbólico quebrado; outros erros (como erros de permissão) são propagados.

Path.**is_block_device**()
Retorna `True` se o caminho apontar para um dispositivo de bloco (ou um link simbólico apontando para um dispositivo de bloco), `False` se apontar para outro tipo de arquivo.

`False` também é retornado se o caminho não existir ou se for um link simbólico quebrado; outros erros (como erros de permissão) são propagados.

Path.**is_char_device**()
Retorna `True` se o caminho apontar para um dispositivo de caractere (ou um link simbólico apontando para um dispositivo de caractere), `False` se apontar para outro tipo de arquivo.

`False` também é retornado se o caminho não existir ou se for um link simbólico quebrado; outros erros (como erros de permissão) são propagados.

Path.**iterdir**()
Quando o caminho aponta para um diretório, produz objetos caminho do conteúdo do diretório:

```
>>> p = Path('docs')
>>> for child in p.iterdir(): child
...
PosixPath('docs/conf.py')
PosixPath('docs/_templates')
PosixPath('docs/make.bat')
PosixPath('docs/index.rst')
PosixPath('docs/_build')
PosixPath('docs/_static')
PosixPath('docs/Makefile')
```

Path.**lchmod**(*mode*)
Como *Path.chmod()*, mas, se o caminho apontar para um link simbólico, o modo do link simbólico é alterado ao invés de seu alvo.

Path.**lstat**()
Como *Path.stat()*, mas, se o caminho apontar para um link simbólico, retorna as informações do link simbólico ao invés de seu alvo.

Path.**mkdir**(*mode=0o777*, *parents=False*, *exist_ok=False*)
Cria um novo diretório neste caminho fornecido. Se *mode* for fornecido, ele é combinado com o valor `umask` do processo para determinar o modo do arquivo e os sinalizadores de acesso. Se o caminho já existe, *FileExistsError* é levantada.

Se *parents* for verdadeiro, quaisquer pais ausentes neste caminho serão criados conforme necessário; eles são criados com as permissões padrão sem levar o *mode* em consideração (imitando o comando POSIX `mkdir -p`).

Se *parents* for falso (o padrão), um pai ausente levanta *FileNotFoundError*.

Se *exist_ok* for falso (o padrão), *FileExistsError* será levantada se o diretório alvo já existir.

Se *exist_ok* for verdadeiro, as exceções *FileExistsError* serão ignoradas (mesmo comportamento que o comando POSIX `mkdir -p`), mas apenas se o último componente do caminho não for um arquivo existente não pertencente ao diretório.

Alterado na versão 3.5: O parâmetro *exist_ok* foi adicionado.

`Path.`**`open`**(*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)

Abre o arquivo apontado pelo caminho, como a função embutida *open()* faz:

```
>>> p = Path('setup.py')
>>> with p.open() as f:
...     f.readline()
...
'#!/usr/bin/env python3\n'
```

`Path.`**`owner`**()

Retorna o nome do usuário que possui o arquivo. *KeyError* é levantada se o uid do arquivo não for encontrado no banco de dados do sistema.

`Path.`**`read_bytes`**()

Retorna o conteúdo binário do arquivo apontado como um objeto bytes:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

Novo na versão 3.5.

`Path.`**`read_text`**(*encoding=None*, *errors=None*)

Retorna o conteúdo decodificado do arquivo apontado como uma string:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

O arquivo é aberto e, então, fechado. Os parâmetros opcionais têm o mesmo significado que em *open()*.

Novo na versão 3.5.

`Path.`**`rename`**(*target*)

Rename this file or directory to the given *target*. On Unix, if *target* exists and is a file, it will be replaced silently if the user has permission. *target* can be either a string or another path object:

```
>>> p = Path('foo')
>>> p.open('w').write('some text')
9
>>> target = Path('bar')
>>> p.rename(target)
>>> target.open().read()
'some text'
```

Path.**replace**(*target*)
Rename this file or directory to the given *target*. If *target* points to an existing file or directory, it will be unconditionally replaced.

Path.**resolve**(*strict=False*)
Faça o caminho absoluto, resolvendo quaisquer links simbólicos. Um novo objeto de caminho é retornado:

```
>>> p = Path()
>>> p
PosixPath('.')
>>> p.resolve()
PosixPath('/home/antoine/pathlib')
```

Componentes "`..`" também são eliminados (este é o único método para fazer isso):

```
>>> p = Path('docs/../setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
```

Se o caminho não existe e *strict* é `True`, *FileNotFoundError* é levantada. Se *strict* for `False`, o caminho será resolvido tanto quanto possível e qualquer resto é anexado sem verificar se existe. Se um laço infinito for encontrado ao longo do caminho de resolução, *RuntimeError* é levantada.

Novo na versão 3.6: O argumento *strict* (comportamento pré-3.6 é estrito).

Path.**rglob**(*pattern*)
É como chamar *Path.glob()* com "`**/`" adicionado na frente do *pattern* relativo fornecido:

```
>>> sorted(Path().rglob("*.py"))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

Path.**rmdir**()
Remove este diretório. O diretório deve estar vazio.

Path.**samefile**(*other_path*)
Retorna se este path apontar para o mesmo arquivo como *other_path*, que pode ser um objeto PATH ou uma String. A semântica é semelhante a função *os.path.samefile()* e a função *os.path.samestat()*.

Um *OSError* poderá ser levantado caso algum arquivo não puder ser acessado por alguma razão.

```
>>> p = Path('spam')
>>> q = Path('eggs')
>>> p.samefile(q)
False
>>> p.samefile('spam')
True
```

Novo na versão 3.5.

Path.**symlink_to**(*target*, *target_is_directory=False*)
Faz deste path um link simbólico para *target*. No Windows, *target_is_directory* deverá ser verdadeiro (padrão `False`) se o local do link for um diretório. Num sistema POSIX, o valor *target_is_directory* será ignorado.

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
```

(continua na próxima página)

(continuação da página anterior)

```
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
>>> p.stat().st_size
956
>>> p.lstat().st_size
8
```

Nota: A ordem dos argumentos (link, target) é o inverso da função *os.symlink()*'s.

`Path.`**`touch`**(*mode=0o666*, *exist_ok=True*)

Cria um arquivo neste caminho especifico. Caso o *modo* for dado, ele será combinado com o valor do processo `umask` para determinar o modo de arquivo e as flags de acesso. Se o arquivo já existir, a função será bem-sucedida se *exist_ok* for verdadeiro (e o tempo de modificação for atualizado para a hora atual), caso contrário o erro *FileExistsError* será levantado.

`Path.`**`unlink`**()

Remova esse arquivo ou link simbólico. Caso o caminho aponte para um diretório, use a função :func:'Path.rmdir" em vez disso.

`Path.`**`write_bytes`**(*data*)

Abre o arquivo apontado no modo bytes, escreve *dados* e fecha o arquivo:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

Um arquivo existente de mesmo nome será substituído.

Novo na versão 3.5.

`Path.`**`write_text`**(*data*, *encoding=None*, *errors=None*)

Abre o arquivo apontado no modo de texto, escreve *data* e fecha o arquivo:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

Novo na versão 3.5.

11.1.4 Correspondência a ferramentas no módulo `os`

Abaixo está uma tabela mapeando várias funções *os* a sua *PurePath*/*Path* equivalente.

Nota: Embora *os.path.relpath()* e *PurePath.relative_to()* tenham alguns casos de uso sobrepostos, sua semântica difere o suficiente para garantir não considerá-los equivalentes.

os e os.path	pathlib
`os.path.abspath()`	`Path.resolve()`
`os.chmod()`	`Path.chmod()`
`os.mkdir()`	`Path.mkdir()`
`os.rename()`	`Path.rename()`
`os.replace()`	`Path.replace()`
`os.rmdir()`	`Path.rmdir()`
`os.remove()`, `os.unlink()`	`Path.unlink()`
`os.getcwd()`	`Path.cwd()`
`os.path.exists()`	`Path.exists()`
`os.path.expanduser()`	`Path.expanduser()` and `Path.home()`
`os.path.isdir()`	`Path.is_dir()`
`os.path.isfile()`	`Path.is_file()`
`os.path.islink()`	`Path.is_symlink()`
`os.stat()`	`Path.stat()`, `Path.owner()`, `Path.group()`
`os.path.isabs()`	`PurePath.is_absolute()`
`os.path.join()`	`PurePath.joinpath()`
`os.path.basename()`	`PurePath.name`
`os.path.dirname()`	`PurePath.parent`
`os.path.samefile()`	`Path.samefile()`
`os.path.splitext()`	`PurePath.suffix`

11.2 `os.path` — Manipulações comuns de nome nomes de caminhos

Código Fonte: Lib/posixpath.py (for POSIX), Lib/ntpath.py (for Windows NT), and Lib/macpath.py (for Macintosh)

Este módulo implementa algumas funções úteis em nomes de caminho. Para ler ou escrever arquivos, veja `open()`, e para acessar o sistema de arquivos veja o módulo `os`. Os parâmetros de caminho podem ser passados como strings ou bytes. As aplicações são encorajadas a representar nomes de arquivos como strings de caracteres (Unicode). Infelizmente, alguns nomes de arquivo podem não ser representados como strings no Unix, então as aplicações que precisam ter suporte a nomes de arquivo arbitrários no Unix devem usar objetos bytes para representar nomes de caminho. Vice-versa, usar objetos bytes não pode representar todos os nomes de arquivos no Windows (na codificação `mbcs` padrão), portanto, as aplicações do Windows devem usar objetos string para acessar todos os arquivos.

Ao contrário de um shell Unix, Python não faz nenhuma expansão *automática* de caminho. Funções como `expanduser()` e `expandvars()` podem ser invocadas explicitamente quando uma aplicação deseja uma expansão de caminho no estilo do shell. (Veja também o módulo `glob`.)

Ver também:

O módulo `pathlib` oferece objetos de caminho de alto nível.

Nota: Todas essas funções aceitam apenas bytes ou apenas objetos de string como seus parâmetros. O resultado é um objeto do mesmo tipo, se um caminho ou nome de arquivo for retornado.

Nota: Uma vez que diferentes sistemas operacionais têm diferentes convenções de nome de caminho, existem várias versões deste módulo na biblioteca padrão. O módulo `os.path` é sempre o módulo de caminho adequado para o sistema

operacional em que o Python está sendo executado e, portanto, pode ser usado para caminhos locais. No entanto, você também pode importar e usar os módulos individuais se quiser manipular um caminho que esteja *sempre* em um dos diferentes formatos. Todos eles têm a mesma interface:

- `posixpath` para caminhos no estilo UNIX
- `ntpath` para caminhos do Windows
- *`macpath`* for old-style MacOS paths

`os.path.`**`abspath`**(*path*)
Retorna uma versão normalizada e absolutizada do nome de caminho *path*. Na maioria das plataformas, isso é equivalente a chamar a função *`normpath()`* da seguinte forma: `normpath(join(os.getcwd(), path))`.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.path.`**`basename`**(*path*)
Retorna o nome base do caminho *path*. Este é o segundo elemento do par retornado pela passagem de *path* para a função *`split()`*. Observe que o resultado desta função é diferente do programa Unix **`basename`**; onde **`basename`** para `'/foo/bar/'` retorna `'bar'`, a função *`basename()`* retorna uma string vazia (`''`).

Alterado na versão 3.6: Aceita um *path-like object*.

`os.path.`**`commonpath`**(*paths*)
Return the longest common sub-path of each pathname in the sequence *paths*. Raise *`ValueError`* if *paths* contains both absolute and relative pathnames, or if *paths* is empty. Unlike *`commonprefix()`*, this returns a valid path.

Disponibilidade: Unix, Windows.

Novo na versão 3.5.

Alterado na versão 3.6: Aceita uma sequência de *objetos caminho ou similar*.

`os.path.`**`commonprefix`**(*list*)
Retorna o prefixo de caminho mais longo (obtido caractere por caractere) que é um prefixo de todos os caminhos em *list*. Se *list* estiver vazia, retorna a string vazia (`''`).

Nota: Esta função pode retornar caminhos inválidos porque funciona um caractere por vez. Para obter um caminho válido, consulte *`commonpath()`*.

```
>>> os.path.commonprefix(['/usr/lib', '/usr/local/lib'])
'/usr/l'

>>> os.path.commonpath(['/usr/lib', '/usr/local/lib'])
'/usr'
```

Alterado na versão 3.6: Aceita um *path-like object*.

`os.path.`**`dirname`**(*path*)
Retorna o nome do diretório do nome de caminho *path*. Este é o primeiro elemento do par retornado passando *path* para a função *`split()`*.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.path.`**`exists`**(*path*)
Retorna `True` se *path* se referir a um caminho existente ou um descritor de arquivo aberto. Retorna `False` para links simbólicos quebrados. Em algumas plataformas, esta função pode retornar `False` se a permissão não for concedida para executar *`os.stat()`* no arquivo solicitado, mesmo se o *path* existir fisicamente.

Alterado na versão 3.3: *path* agora pode ser um inteiro: `True` é retornado se for um descritor de arquivo aberto, `False` caso contrário.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.path.`**`lexists`**(*path*)
Retorna `True` se *path* se referir a um caminho existente. Retorna `True` para links simbólicos quebrados. Equivalente a `exists()` em plataformas sem `os.lstat()`.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.path.`**`expanduser`**(*path*)
No Unix e no Windows, retorna o argumento com um componente inicial de ~ ou `~user` substituído pelo diretório inicial daquele usuário *user*.

No Unix, um ~ no início é substituído pela variável de ambiente `HOME` se estiver definida; caso contrário, o diretório pessoal do usuário atual é procurado no diretório de senha através do módulo embutido `pwd`. Um `~user` no início é procurado diretamente no diretório de senhas.

On Windows, `HOME` and `USERPROFILE` will be used if set, otherwise a combination of `HOMEPATH` and `HOMEDRIVE` will be used. An initial `~user` is handled by stripping the last directory component from the created user path derived above.

Se a expansão falhar ou se o caminho não começar com um til, o caminho será retornado inalterado.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.path.`**`expandvars`**(*path*)
Retorna o argumento com as variáveis de ambiente expandidas. Substrings da forma `$name` ou `${name}` são substituídas pelo valor da variável de ambiente *name*. Nomes de variáveis malformados e referências a variáveis não existentes permanecem inalterados.

No Windows, expansões `%name%` são suportadas juntamente a `$name` e `${name}`.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.path.`**`getatime`**(*path*)
Retorna a hora do último acesso de *path*. O valor de retorno é um número de ponto flutuante dando o número de segundos desde a Era Unix (veja o módulo `time`). Levanta `OSError` se o arquivo não existe ou está inacessível.

`os.path.`**`getmtime`**(*path*)
Retorna a hora da última modificação de *path*. O valor de retorno é um número de ponto flutuante dando o número de segundos desde a Era Unix (veja o módulo `time`). Levanta `OSError` se o arquivo não existe ou está inacessível.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.path.`**`getctime`**(*path*)
Retorna o ctime do sistema que, em alguns sistemas (como Unix) é a hora da última alteração de metadados, e, em outros (como Windows), é a hora de criação de *path*. O valor de retorno é um número que fornece o número de segundos desde a Era Unix (veja o módulo `time`). Levanta `OSError` se o arquivo não existe ou está inacessível.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.path.`**`getsize`**(*path*)
Retorna o tamanho, em bytes, de *path*. Levanta `OSError` se o arquivo não existe ou está inacessível.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.path.`**`isabs`**(*path*)
Retorna `True` se *path* for um nome de caminho absoluto. No Unix, isso significa que começa com uma barra, no Windows começa com uma barra (invertida) depois de eliminar uma possível letra de unidade.

Alterado na versão 3.6: Aceita um *path-like object*.

os.path.**isfile**(*path*)
Retorna `True` se *path* for um arquivo regular *existente*. Isso segue links simbólicos, então *islink()* e *isfile()* podem ser verdadeiros para o mesmo caminho.

Alterado na versão 3.6: Aceita um *path-like object*.

os.path.**isdir**(*path*)
Retorna `True` se *path* for um diretório *existente*. Isso segue links simbólicos, então *islink()* e *isdir()* podem ser verdadeiros para o mesmo caminho.

Alterado na versão 3.6: Aceita um *path-like object*.

os.path.**islink**(*path*)
Retorna `True` se *path* se referir a uma entrada de diretório *existente* que é um link simbólico. Sempre `False` se links simbólicos não forem suportados pelo tempo de execução Python.

Alterado na versão 3.6: Aceita um *path-like object*.

os.path.**ismount**(*path*)
Retorna `True` se o nome de caminho *path* for um *ponto de montagem*: um ponto em um sistema de arquivos onde um sistema de arquivos diferente foi montado. No POSIX, a função verifica se o pai de *path*, *path*`/..`, está em um dispositivo diferente de *path*, ou se *path*`/..` e *path* apontam para o mesmo nó-i no mesmo dispositivo – isso deve detectar pontos de montagem para todas as variantes Unix e POSIX. Não é capaz de detectar confiavelmente montagens bind no mesmo sistema de arquivos. No Windows, uma raiz de letra de unidade e um UNC de compartilhamento são sempre pontos de montagem e, para qualquer outro caminho, `GetVolumePathName` é chamado para ver se é diferente do caminho de entrada.

Novo na versão 3.4: Suporte para detecção de pontos de montagem não raiz no Windows.

Alterado na versão 3.6: Aceita um *path-like object*.

os.path.**join**(*path*, **paths*)
Junta um ou mais componentes do caminho de forma inteligente. O valor de retorno é a concatenação de *path* e qualquer membro de **paths* com exatamente um separador de diretório (`os.sep`) seguindo cada parte não vazia exceto a última, o que significa que o resultado só terminará em um separador se a última parte estiver vazia. Se um componente for um caminho absoluto, todos os componentes anteriores serão descartados e a união continuará a partir do componente do caminho absoluto.

No Windows, a letra da unidade não é redefinida quando um componente de caminho absoluto (por exemplo, `r'\foo'`) é encontrado. Se um componente contiver uma letra de unidade, todos os componentes anteriores serão descartados e a letra da unidade será redefinida. Observe que, como há um diretório atual para cada unidade, `os.path.join("c:", "foo")` representa um caminho relativo ao diretório atual na unidade `C:` (`c:foo`), e não `c:\foo`.

Alterado na versão 3.6: Aceita um *objeto caminho ou similar* para *path* e *paths*.

os.path.**normcase**(*path*)
Normalize the case of a pathname. On Windows, convert all characters in the pathname to lowercase, and also convert forward slashes to backward slashes. On other operating systems, return the path unchanged. Raise a *TypeError* if the type of *path* is not `str` or `bytes` (directly or indirectly through the *os.PathLike* interface).

Alterado na versão 3.6: Aceita um *path-like object*.

os.path.**normpath**(*path*)
Normaliza um nome de caminho retirando separadores redundantes e referências de nível superior para que `A//B`, `A/B/`, `A/./B` e `A/foo/../B` todos se tornem `A/B`. Essa manipulação de string pode mudar o significado de um caminho que contém links simbólicos. No Windows, ele converte barras normais em barras invertidas. Para normalizar o estado de letras maiúsculas/minúsculas, use *normcase()*.

Alterado na versão 3.6: Aceita um *path-like object*.

os.path.**realpath**(*path*)
Retorna o caminho canônico do nome do arquivo especificado, eliminando quaisquer links simbólicos encontrados no caminho (se esses forem suportados pelo sistema operacional).

Alterado na versão 3.6: Aceita um *path-like object*.

os.path.**relpath**(*path*, *start=os.curdir*)
Retorna um caminho de arquivo relativo a *caminho* do diretório atual ou de um diretório *start* opcional. Este é um cálculo de caminho: o sistema de arquivos não é acessado para confirmar a existência ou natureza de *path* ou *start*.

start tem como padrão `os.curdir`.

Disponibilidade: Unix, Windows.

Alterado na versão 3.6: Aceita um *path-like object*.

os.path.**samefile**(*path1*, *path2*)
Retorna `True` se ambos os argumentos de nome de caminho se referem ao mesmo arquivo ou diretório. Isso é determinado pelo número do dispositivo e número do nó-i e levanta uma exceção se uma chamada `os.stat()` em qualquer um dos caminhos falhar.

Disponibilidade: Unix, Windows.

Alterado na versão 3.2: Adicionado suporte ao Windows.

Alterado na versão 3.4: O Windows agora usa a mesma implementação que todas as outras plataformas.

Alterado na versão 3.6: Aceita um *path-like object*.

os.path.**sameopenfile**(*fp1*, *fp2*)
Retorna `True` se os descritores de arquivo *fp1* e *fp2* fazem referência ao mesmo arquivo.

Disponibilidade: Unix, Windows.

Alterado na versão 3.2: Adicionado suporte ao Windows.

Alterado na versão 3.6: Aceita um *path-like object*.

os.path.**samestat**(*stat1*, *stat2*)
Retorna `True` se as tuplas de estatísticas *stat1* e *stat2* fazem referência ao mesmo arquivo. Essas estruturas podem ter sido retornadas por `os.fstat()`, `os.lstat()` ou `os.stat()`. Esta função implementa a comparação subjacente usada por `samefile()` e `sameopenfile()`.

Disponibilidade: Unix, Windows.

Alterado na versão 3.4: Adicionado suporte ao Windows.

Alterado na versão 3.6: Aceita um *path-like object*.

os.path.**split**(*path*)
Divide o caminho *path* em um par, `(cabeça, rabo)` onde *rabo* é o último componente do nome do caminho e *cabeça* é tudo o que leva a isso. A parte *rabo* nunca conterá uma barra; se *path* terminar com uma barra, *tail* ficará vazio. Se não houver uma barra no *path*, o *head* ficará vazio. Se *path* estiver vazio, *cabeça* e *rabo* estarão vazios. As barras finais são retiradas da *cabeça*, a menos que seja a raiz (uma ou mais barras apenas). Em todos os casos, `join(cabeça, rabo)` retorna um caminho para o mesmo local que *path* (mas as strings podem ser diferentes). Veja também as funções `dirname()` e `basename()`.

Alterado na versão 3.6: Aceita um *path-like object*.

os.path.**splitdrive**(*path*)
Divide o nome do caminho *path* em um par `(unidade, rabo)` onde *unidade* é um ponto de montagem ou uma string vazia. Em sistemas que não usam especificações de unidade, *unidade* sempre será a string vazia. Em todos os casos, `unidade + rabo` será o mesmo que *path*.

No Windows, divide um nome de caminho em unidade/ponto de compartilhamento UNC e caminho relativo.

Se o caminho contiver uma letra de unidade, a unidade conterá tudo, até e incluindo os dois pontos, por exemplo, `splitdrive("c:/dir")` retorna `("c:", "/dir")`

Se o caminho contiver um caminho UNC, a unidade conterá o nome do host e o compartilhamento, até mas não incluindo o quarto separador. por exemplo, `splitdrive("//host/computer/dir")` retorna `("//host/computer", "/dir")`

Alterado na versão 3.6: Aceita um *path-like object*.

`os.path.`**`splitext`**(*path*)
Divide o nome de caminho *path* em um par `(root, ext)` de tal forma que `root + ext == path`, e *ext* esteja vazio ou inicia com um ponto e contém no máximo um ponto. Pontos no início do nome base são ignorados; `splitext('.cshrc')` retorna `('.cshrc', '')`.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.path.`**`supports_unicode_filenames`**
`True` se strings Unicode arbitrárias podem ser usadas como nomes de arquivo (dentro das limitações impostas pelo sistema de arquivos).

11.3 `fileinput` — Iterate over lines from multiple input streams

Código Fonte: Lib/fileinput.py

This module implements a helper class and functions to quickly write a loop over standard input or a list of files. If you just want to read or write one file see *`open()`*.

O uso típico é:

```
import fileinput
for line in fileinput.input():
    process(line)
```

This iterates over the lines of all files listed in `sys.argv[1:]`, defaulting to `sys.stdin` if the list is empty. If a filename is `'-'`, it is also replaced by `sys.stdin` and the optional arguments *mode* and *openhook* are ignored. To specify an alternative list of filenames, pass it as the first argument to *`input()`*. A single file name is also allowed.

All files are opened in text mode by default, but you can override this by specifying the *mode* parameter in the call to *`input()`* or *`FileInput`*. If an I/O error occurs during opening or reading a file, *`OSError`* is raised.

Alterado na versão 3.3: *`IOError`* used to be raised; it is now an alias of *`OSError`*.

If `sys.stdin` is used more than once, the second and further use will return no lines, except perhaps for interactive use, or if it has been explicitly reset (e.g. using `sys.stdin.seek(0)`).

Empty files are opened and immediately closed; the only time their presence in the list of filenames is noticeable at all is when the last file opened is empty.

Lines are returned with any newlines intact, which means that the last line in a file may not have one.

You can control how files are opened by providing an opening hook via the *openhook* parameter to *`fileinput.input()`* or *`FileInput()`*. The hook must be a function that takes two arguments, *filename* and *mode*, and returns an accordingly opened file-like object. Two useful hooks are already provided by this module.

The following function is the primary interface of this module:

fileinput.**input**(*files=None*, *inplace=False*, *backup=''*, *bufsize=0*, *mode='r'*, *openhook=None*)
Create an instance of the *FileInput* class. The instance will be used as global state for the functions of this module, and is also returned to use during iteration. The parameters to this function will be passed along to the constructor of the *FileInput* class.

The *FileInput* instance can be used as a context manager in the `with` statement. In this example, *input* is closed after the `with` statement is exited, even if an exception occurs:

```
with fileinput.input(files=('spam.txt', 'eggs.txt')) as f:
    for line in f:
        process(line)
```

Alterado na versão 3.2: Can be used as a context manager.

Deprecated since version 3.6, will be removed in version 3.8: The *bufsize* parameter.

The following functions use the global state created by *fileinput.input()*; if there is no active state, *RuntimeError* is raised.

fileinput.**filename**()
Return the name of the file currently being read. Before the first line has been read, returns `None`.

fileinput.**fileno**()
Return the integer "file descriptor" for the current file. When no file is opened (before the first line and between files), returns `-1`.

fileinput.**lineno**()
Return the cumulative line number of the line that has just been read. Before the first line has been read, returns `0`. After the last line of the last file has been read, returns the line number of that line.

fileinput.**filelineno**()
Return the line number in the current file. Before the first line has been read, returns `0`. After the last line of the last file has been read, returns the line number of that line within the file.

fileinput.**isfirstline**()
Return `True` if the line just read is the first line of its file, otherwise return `False`.

fileinput.**isstdin**()
Return `True` if the last line was read from `sys.stdin`, otherwise return `False`.

fileinput.**nextfile**()
Close the current file so that the next iteration will read the first line from the next file (if any); lines not read from the file will not count towards the cumulative line count. The filename is not changed until after the first line of the next file has been read. Before the first line has been read, this function has no effect; it cannot be used to skip the first file. After the last line of the last file has been read, this function has no effect.

fileinput.**close**()
Close the sequence.

The class which implements the sequence behavior provided by the module is available for subclassing as well:

class fileinput.**FileInput**(*files=None*, *inplace=False*, *backup=''*, *bufsize=0*, *mode='r'*, *openhook=None*)
Class *FileInput* is the implementation; its methods *filename()*, *fileno()*, *lineno()*, *filelineno()*, *isfirstline()*, *isstdin()*, *nextfile()* and *close()* correspond to the functions of the same name in the module. In addition it has a *readline()* method which returns the next input line, and a `__getitem__()` method which implements the sequence behavior. The sequence must be accessed in strictly sequential order; random access and *readline()* cannot be mixed.

With *mode* you can specify which file mode will be passed to *open()*. It must be one of `'r'`, `'rU'`, `'U'` and `'rb'`.

The *openhook*, when given, must be a function that takes two arguments, *filename* and *mode*, and returns an accordingly opened file-like object. You cannot use *inplace* and *openhook* together.

A *FileInput* instance can be used as a context manager in the `with` statement. In this example, *input* is closed after the `with` statement is exited, even if an exception occurs:

```
with FileInput(files=('spam.txt', 'eggs.txt')) as input:
    process(input)
```

Alterado na versão 3.2: Can be used as a context manager.

Obsoleto desde a versão 3.4: The `'rU'` and `'U'` modes.

Deprecated since version 3.6, will be removed in version 3.8: The *bufsize* parameter.

Optional in-place filtering: if the keyword argument `inplace=True` is passed to *fileinput.input()* or to the *FileInput* constructor, the file is moved to a backup file and standard output is directed to the input file (if a file of the same name as the backup file already exists, it will be replaced silently). This makes it possible to write a filter that rewrites its input file in place. If the *backup* parameter is given (typically as `backup='.<some extension>'`), it specifies the extension for the backup file, and the backup file remains around; by default, the extension is `'.bak'` and it is deleted when the output file is closed. In-place filtering is disabled when standard input is read.

The two following opening hooks are provided by this module:

`fileinput.`**`hook_compressed`**(*filename*, *mode*)
Transparently opens files compressed with gzip and bzip2 (recognized by the extensions `'.gz'` and `'.bz2'`) using the *gzip* and *bz2* modules. If the filename extension is not `'.gz'` or `'.bz2'`, the file is opened normally (ie, using *open()* without any decompression).

Usage example: `fi = fileinput.FileInput(openhook=fileinput.hook_compressed)`

`fileinput.`**`hook_encoded`**(*encoding*, *errors=None*)
Returns a hook which opens each file with *open()*, using the given *encoding* and *errors* to read the file.

Usage example: `fi = fileinput.FileInput(openhook=fileinput.hook_encoded("utf-8", "surrogateescape"))`

Alterado na versão 3.6: Added the optional *errors* parameter.

11.4 `stat` — Interpreting `stat()` results

Código Fonte: Lib/stat.py

O módulo *stat* define constantes e funções para interpretar os resultados de *os.stat()*, *os.fstat()* e *os.lstat()* (se existirem). Para detalhes completos sobre chamadas `stat()`, `fstat()` e `lstat()`, consulte a documentação do seu sistema.

Alterado na versão 3.4: The stat module is backed by a C implementation.

The *stat* module defines the following functions to test for specific file types:

`stat.`**`S_ISDIR`**(*mode*)
Return non-zero if the mode is from a directory.

`stat.`**`S_ISCHR`**(*mode*)
Return non-zero if the mode is from a character special device file.

`stat.`**`S_ISBLK`**(*mode*)
Return non-zero if the mode is from a block special device file.

stat.**S_ISREG**(*mode*)
Return non-zero if the mode is from a regular file.

stat.**S_ISFIFO**(*mode*)
Return non-zero if the mode is from a FIFO (named pipe).

stat.**S_ISLNK**(*mode*)
Return non-zero if the mode is from a symbolic link.

stat.**S_ISSOCK**(*mode*)
Return non-zero if the mode is from a socket.

stat.**S_ISDOOR**(*mode*)
Return non-zero if the mode is from a door.

Novo na versão 3.4.

stat.**S_ISPORT**(*mode*)
Return non-zero if the mode is from an event port.

Novo na versão 3.4.

stat.**S_ISWHT**(*mode*)
Return non-zero if the mode is from a whiteout.

Novo na versão 3.4.

Two additional functions are defined for more general manipulation of the file's mode:

stat.**S_IMODE**(*mode*)
Return the portion of the file's mode that can be set by *os.chmod()*—that is, the file's permission bits, plus the sticky bit, set-group-id, and set-user-id bits (on systems that support them).

stat.**S_IFMT**(*mode*)
Return the portion of the file's mode that describes the file type (used by the `S_IS*()` functions above).

Normally, you would use the `os.path.is*()` functions for testing the type of a file; the functions here are useful when you are doing multiple tests of the same file and wish to avoid the overhead of the `stat()` system call for each test. These are also useful when checking for information about a file that isn't handled by *os.path*, like the tests for block and character devices.

Exemplo:

```
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top,
       calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.stat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print('Skipping %s' % pathname)
```

(continua na próxima página)

(continuação da página anterior)

```
def visitfile(file):
    print('visiting', file)

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

An additional utility function is provided to convert a file's mode in a human readable string:

stat.**filemode**(*mode*)
Convert a file's mode to a string of the form '-rwxrwxrwx'.

Novo na versão 3.3.

Alterado na versão 3.4: The function supports *S_IFDOOR*, *S_IFPORT* and *S_IFWHT*.

All the variables below are simply symbolic indexes into the 10-tuple returned by *os.stat()*, *os.fstat()* or *os.lstat()*.

stat.**ST_MODE**
Inode protection mode.

stat.**ST_INO**
Inode number.

stat.**ST_DEV**
Device inode resides on.

stat.**ST_NLINK**
Number of links to the inode.

stat.**ST_UID**
User id of the owner.

stat.**ST_GID**
Group id of the owner.

stat.**ST_SIZE**
Size in bytes of a plain file; amount of data waiting on some special files.

stat.**ST_ATIME**
Time of last access.

stat.**ST_MTIME**
Time of last modification.

stat.**ST_CTIME**
The "ctime" as reported by the operating system. On some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time (see platform documentation for details).

The interpretation of "file size" changes according to the file type. For plain files this is the size of the file in bytes. For FIFOs and sockets under most flavors of Unix (including Linux in particular), the "size" is the number of bytes waiting to be read at the time of the call to *os.stat()*, *os.fstat()*, or *os.lstat()*; this can sometimes be useful, especially for polling one of these special files after a non-blocking open. The meaning of the size field for other character and block devices varies more, depending on the implementation of the underlying system call.

The variables below define the flags used in the *ST_MODE* field.

Use of the functions above is more portable than use of the first set of flags:

stat.**S_IFSOCK**
Socket.

stat.**S_IFLNK**
Symbolic link.

stat.**S_IFREG**
Regular file.

stat.**S_IFBLK**
Block device.

stat.**S_IFDIR**
Directory.

stat.**S_IFCHR**
Character device.

stat.**S_IFIFO**
FIFO.

stat.**S_IFDOOR**
Door.

Novo na versão 3.4.

stat.**S_IFPORT**
Event port.

Novo na versão 3.4.

stat.**S_IFWHT**
Whiteout.

Novo na versão 3.4.

Nota: *S_IFDOOR*, *S_IFPORT* or *S_IFWHT* are defined as 0 when the platform does not have support for the file types.

The following flags can also be used in the *mode* argument of *os.chmod()*:

stat.**S_ISUID**
Set UID bit.

stat.**S_ISGID**
Set-group-ID bit. This bit has several special uses. For a directory it indicates that BSD semantics is to be used for that directory: files created there inherit their group ID from the directory, not from the effective group ID of the creating process, and directories created there will also get the *S_ISGID* bit set. For a file that does not have the group execution bit (*S_IXGRP*) set, the set-group-ID bit indicates mandatory file/record locking (see also *S_ENFMT*).

stat.**S_ISVTX**
Sticky bit. When this bit is set on a directory it means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, or by a privileged process.

stat.**S_IRWXU**
Mask for file owner permissions.

stat.**S_IRUSR**
Owner has read permission.

stat.**S_IWUSR**
Owner has write permission.

stat.**S_IXUSR**
Owner has execute permission.

stat.**S_IRWXG**
Mask for group permissions.

stat.**S_IRGRP**
Group has read permission.

stat.**S_IWGRP**
Group has write permission.

stat.**S_IXGRP**
Grupo tem permissão de execução.

stat.**S_IRWXO**
Mask for permissions for others (not in group).

stat.**S_IROTH**
Others have read permission.

stat.**S_IWOTH**
Others have write permission.

stat.**S_IXOTH**
Others have execute permission.

stat.**S_ENFMT**
System V file locking enforcement. This flag is shared with *S_ISGID*: file/record locking is enforced on files that do not have the group execution bit (*S_IXGRP*) set.

stat.**S_IREAD**
Unix V7 synonym for *S_IRUSR*.

stat.**S_IWRITE**
Unix V7 synonym for *S_IWUSR*.

stat.**S_IEXEC**
Unix V7 synonym for *S_IXUSR*.

The following flags can be used in the *flags* argument of *os.chflags()*:

stat.**UF_NODUMP**
Do not dump the file.

stat.**UF_IMMUTABLE**
The file may not be changed.

stat.**UF_APPEND**
The file may only be appended to.

stat.**UF_OPAQUE**
The directory is opaque when viewed through a union stack.

stat.**UF_NOUNLINK**
The file may not be renamed or deleted.

stat.**UF_COMPRESSED**
The file is stored compressed (Mac OS X 10.6+).

stat.**UF_HIDDEN**
The file should not be displayed in a GUI (Mac OS X 10.5+).

stat.**SF_ARCHIVED**
The file may be archived.

stat.**SF_IMMUTABLE**
The file may not be changed.

stat.**SF_APPEND**
The file may only be appended to.

stat.**SF_NOUNLINK**
The file may not be renamed or deleted.

stat.**SF_SNAPSHOT**
The file is a snapshot file.

See the *BSD or Mac OS systems man page *chflags(2)* for more information.

On Windows, the following file attribute constants are available for use when testing bits in the st_file_attributes member returned by *os.stat()*. See the Windows API documentation for more detail on the meaning of these constants.

stat.**FILE_ATTRIBUTE_ARCHIVE**
stat.**FILE_ATTRIBUTE_COMPRESSED**
stat.**FILE_ATTRIBUTE_DEVICE**
stat.**FILE_ATTRIBUTE_DIRECTORY**
stat.**FILE_ATTRIBUTE_ENCRYPTED**
stat.**FILE_ATTRIBUTE_HIDDEN**
stat.**FILE_ATTRIBUTE_INTEGRITY_STREAM**
stat.**FILE_ATTRIBUTE_NORMAL**
stat.**FILE_ATTRIBUTE_NOT_CONTENT_INDEXED**
stat.**FILE_ATTRIBUTE_NO_SCRUB_DATA**
stat.**FILE_ATTRIBUTE_OFFLINE**
stat.**FILE_ATTRIBUTE_READONLY**
stat.**FILE_ATTRIBUTE_REPARSE_POINT**
stat.**FILE_ATTRIBUTE_SPARSE_FILE**
stat.**FILE_ATTRIBUTE_SYSTEM**
stat.**FILE_ATTRIBUTE_TEMPORARY**
stat.**FILE_ATTRIBUTE_VIRTUAL**
Novo na versão 3.5.

11.5 filecmp — Comparações de arquivos e diretórios

Código Fonte: Lib/filecmp.py

The *filecmp* module defines functions to compare files and directories, with various optional time/correctness trade-offs. For comparing files, see also the *difflib* module.

The *filecmp* module defines the following functions:

filecmp.**cmp**(*f1*, *f2*, *shallow=True*)
Compare the files named *f1* and *f2*, returning True if they seem equal, False otherwise.

If *shallow* is true, files with identical *os.stat()* signatures are taken to be equal. Otherwise, the contents of the files are compared.

Note that no external programs are called from this function, giving it portability and efficiency.

This function uses a cache for past comparisons and the results, with cache entries invalidated if the *os.stat()* information for the file changes. The entire cache may be cleared using *clear_cache()*.

filecmp.**cmpfiles**(*dir1*, *dir2*, *common*, *shallow=True*)
Compare the files in the two directories *dir1* and *dir2* whose names are given by *common*.

Returns three lists of file names: *match*, *mismatch*, *errors*. *match* contains the list of files that match, *mismatch* contains the names of those that don't, and *errors* lists the names of files which could not be compared. Files are listed in *errors* if they don't exist in one of the directories, the user lacks permission to read them or if the comparison could not be done for some other reason.

The *shallow* parameter has the same meaning and default value as for *filecmp.cmp()*.

For example, `cmpfiles('a', 'b', ['c', 'd/e'])` will compare `a/c` with `b/c` and `a/d/e` with `b/d/e`. `'c'` and `'d/e'` will each be in one of the three returned lists.

filecmp.**clear_cache**()
Clear the filecmp cache. This may be useful if a file is compared so quickly after it is modified that it is within the mtime resolution of the underlying filesystem.

Novo na versão 3.4.

11.5.1 A classe `dircmp`

class filecmp.**dircmp**(*a*, *b*, *ignore=None*, *hide=None*)
Construct a new directory comparison object, to compare the directories *a* and *b*. *ignore* is a list of names to ignore, and defaults to *filecmp.DEFAULT_IGNORES*. *hide* is a list of names to hide, and defaults to `[os.curdir, os.pardir]`.

The *dircmp* class compares files by doing *shallow* comparisons as described for *filecmp.cmp()*.

The *dircmp* class provides the following methods:

report()
Print (to *sys.stdout*) a comparison between *a* and *b*.

report_partial_closure()
Print a comparison between *a* and *b* and common immediate subdirectories.

report_full_closure()
Print a comparison between *a* and *b* and common subdirectories (recursively).

The *dircmp* class offers a number of interesting attributes that may be used to get various bits of information about the directory trees being compared.

Note that via `__getattr__()` hooks, all attributes are computed lazily, so there is no speed penalty if only those attributes which are lightweight to compute are used.

left
The directory *a*.

right
The directory *b*.

left_list
Files and subdirectories in *a*, filtered by *hide* and *ignore*.

right_list
Files and subdirectories in *b*, filtered by *hide* and *ignore*.

common
Files and subdirectories in both *a* and *b*.

left_only
Files and subdirectories only in *a*.

right_only
Files and subdirectories only in *b*.

common_dirs
Subdirectories in both *a* and *b*.

common_files
Arquivos em *a* e *b*.

common_funny
Names in both *a* and *b*, such that the type differs between the directories, or names for which *os.stat()* reports an error.

same_files
Files which are identical in both *a* and *b*, using the class's file comparison operator.

diff_files
Files which are in both *a* and *b*, whose contents differ according to the class's file comparison operator.

funny_files
Files which are in both *a* and *b*, but could not be compared.

subdirs
A dictionary mapping names in *common_dirs* to *dircmp* objects.

filecmp.**DEFAULT_IGNORES**
Novo na versão 3.4.

List of directories ignored by *dircmp* by default.

Here is a simplified example of using the `subdirs` attribute to search recursively through two directories to show common different files:

```
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print("diff_file %s found in %s and %s" % (name, dcmp.left,
...               dcmp.right))
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```

11.6 tempfile — Gerar arquivos temporários e diretórios

Código Fonte: Lib/tempfile.py

This module creates temporary files and directories. It works on all supported platforms. *TemporaryFile*, *NamedTemporaryFile*, *TemporaryDirectory*, and *SpooledTemporaryFile* are high-level interfaces which provide automatic cleanup and can be used as context managers. *mkstemp()* and *mkdtemp()* are lower-level functions which require manual cleanup.

All the user-callable functions and constructors take additional arguments which allow direct control over the location and name of temporary files and directories. Files names used by this module include a string of random characters

which allows those files to be securely created in shared temporary directories. To maintain backward compatibility, the argument order is somewhat odd; it is recommended to use keyword arguments for clarity.

The module defines the following user-callable items:

tempfile.**TemporaryFile**(*mode='w+b'*, *buffering=None*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*)

Return a *file-like object* that can be used as a temporary storage area. The file is created securely, using the same rules as `mkstemp()`. It will be destroyed as soon as it is closed (including an implicit close when the object is garbage collected). Under Unix, the directory entry for the file is either not created at all or is removed immediately after the file is created. Other platforms do not support this; your code should not rely on a temporary file created using this function having or not having a visible name in the file system.

The resulting object can be used as a context manager (see *Exemplos*). On completion of the context or destruction of the file object the temporary file will be removed from the filesystem.

The *mode* parameter defaults to `'w+b'` so that the file created can be read and written without being closed. Binary mode is used so that it behaves consistently on all platforms without regard for the data that is stored. *buffering*, *encoding* and *newline* are interpreted as for `open()`.

The *dir*, *prefix* and *suffix* parameters have the same meaning and defaults as with `mkstemp()`.

The returned object is a true file object on POSIX platforms. On other platforms, it is a file-like object whose `file` attribute is the underlying true file object.

The `os.O_TMPFILE` flag is used if it is available and works (Linux-specific, requires Linux kernel 3.11 or later).

Alterado na versão 3.5: The `os.O_TMPFILE` flag is now used if available.

tempfile.**NamedTemporaryFile**(*mode='w+b'*, *buffering=None*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, *delete=True*)

This function operates exactly as `TemporaryFile()` does, except that the file is guaranteed to have a visible name in the file system (on Unix, the directory entry is not unlinked). That name can be retrieved from the `name` attribute of the returned file-like object. Whether the name can be used to open the file a second time, while the named temporary file is still open, varies across platforms (it can be so used on Unix; it cannot on Windows NT or later). If *delete* is true (the default), the file is deleted as soon as it is closed. The returned object is always a file-like object whose `file` attribute is the underlying true file object. This file-like object can be used in a `with` statement, just like a normal file.

tempfile.**SpooledTemporaryFile**(*max_size=0*, *mode='w+b'*, *buffering=None*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*)

This function operates exactly as `TemporaryFile()` does, except that data is spooled in memory until the file size exceeds *max_size*, or until the file's `fileno()` method is called, at which point the contents are written to disk and operation proceeds as with `TemporaryFile()`.

The resulting file has one additional method, `rollover()`, which causes the file to roll over to an on-disk file regardless of its size.

The returned object is a file-like object whose `_file` attribute is either an `io.BytesIO` or `io.TextIOWrapper` object (depending on whether binary or text *mode* was specified) or a true file object, depending on whether `rollover()` has been called. This file-like object can be used in a `with` statement, just like a normal file.

Alterado na versão 3.3: the truncate method now accepts a `size` argument.

tempfile.**TemporaryDirectory**(*suffix=None*, *prefix=None*, *dir=None*)

This function securely creates a temporary directory using the same rules as `mkdtemp()`. The resulting object can be used as a context manager (see *Exemplos*). On completion of the context or destruction of the temporary directory object the newly created temporary directory and all its contents are removed from the filesystem.

The directory name can be retrieved from the `name` attribute of the returned object. When the returned object is used as a context manager, the `name` will be assigned to the target of the `as` clause in the `with` statement, if there

is one.

The directory can be explicitly cleaned up by calling the `cleanup()` method.

Novo na versão 3.2.

`tempfile.`**`mkstemp`**(*suffix=None*, *prefix=None*, *dir=None*, *text=False*)

Creates a temporary file in the most secure manner possible. There are no race conditions in the file's creation, assuming that the platform properly implements the *os.O_EXCL* flag for *os.open()*. The file is readable and writable only by the creating user ID. If the platform uses permission bits to indicate whether a file is executable, the file is executable by no one. The file descriptor is not inherited by child processes.

Unlike *TemporaryFile()*, the user of *mkstemp()* is responsible for deleting the temporary file when done with it.

If *suffix* is not `None`, the file name will end with that suffix, otherwise there will be no suffix. *mkstemp()* does not put a dot between the file name and the suffix; if you need one, put it at the beginning of *suffix*.

If *prefix* is not `None`, the file name will begin with that prefix; otherwise, a default prefix is used. The default is the return value of *gettempprefix()* or *gettempprefixb()*, as appropriate.

If *dir* is not `None`, the file will be created in that directory; otherwise, a default directory is used. The default directory is chosen from a platform-dependent list, but the user of the application can control the directory location by setting the *TMPDIR*, *TEMP* or *TMP* environment variables. There is thus no guarantee that the generated filename will have any nice properties, such as not requiring quoting when passed to external commands via `os.popen()`.

If any of *suffix*, *prefix*, and *dir* are not `None`, they must be the same type. If they are bytes, the returned name will be bytes instead of str. If you want to force a bytes return value with otherwise default behavior, pass `suffix=b''`.

If *text* is specified, it indicates whether to open the file in binary mode (the default) or text mode. On some platforms, this makes no difference.

mkstemp() returns a tuple containing an OS-level handle to an open file (as would be returned by *os.open()*) and the absolute pathname of that file, in that order.

Alterado na versão 3.5: *suffix*, *prefix*, and *dir* may now be supplied in bytes in order to obtain a bytes return value. Prior to this, only str was allowed. *suffix* and *prefix* now accept and default to `None` to cause an appropriate default value to be used.

Alterado na versão 3.6: The *dir* parameter now accepts a *path-like object*.

`tempfile.`**`mkdtemp`**(*suffix=None*, *prefix=None*, *dir=None*)

Creates a temporary directory in the most secure manner possible. There are no race conditions in the directory's creation. The directory is readable, writable, and searchable only by the creating user ID.

The user of *mkdtemp()* is responsible for deleting the temporary directory and its contents when done with it.

The *prefix*, *suffix*, and *dir* arguments are the same as for *mkstemp()*.

mkdtemp() returns the absolute pathname of the new directory.

Alterado na versão 3.5: *suffix*, *prefix*, and *dir* may now be supplied in bytes in order to obtain a bytes return value. Prior to this, only str was allowed. *suffix* and *prefix* now accept and default to `None` to cause an appropriate default value to be used.

Alterado na versão 3.6: The *dir* parameter now accepts a *path-like object*.

`tempfile.`**`gettempdir`**()

Return the name of the directory used for temporary files. This defines the default value for the *dir* argument to all functions in this module.

Python searches a standard list of directories to find one which the calling user can create files in. The list is:

1. The directory named by the `TMPDIR` environment variable.

2. The directory named by the `TEMP` environment variable.
3. The directory named by the `TMP` environment variable.
4. Uma localização específica por plataforma:
   - On Windows, the directories `C:\TEMP`, `C:\TMP`, `\TEMP`, and `\TMP`, in that order.
   - On all other platforms, the directories `/tmp`, `/var/tmp`, and `/usr/tmp`, in that order.
5. As a last resort, the current working directory.

The result of this search is cached, see the description of *tempdir* below.

`tempfile.`**`gettempdirb`**`()`
Same as *gettempdir()* but the return value is in bytes.

Novo na versão 3.5.

`tempfile.`**`gettempprefix`**`()`
Return the filename prefix used to create temporary files. This does not contain the directory component.

`tempfile.`**`gettempprefixb`**`()`
Same as *gettempprefix()* but the return value is in bytes.

Novo na versão 3.5.

The module uses a global variable to store the name of the directory used for temporary files returned by *gettempdir()*. It can be set directly to override the selection process, but this is discouraged. All functions in this module take a *dir* argument which can be used to specify the directory and this is the recommended approach.

`tempfile.`**`tempdir`**
When set to a value other than `None`, this variable defines the default value for the *dir* argument to the functions defined in this module.

If `tempdir` is `None` (the default) at any call to any of the above functions except *gettempprefix()* it is initialized following the algorithm described in *gettempdir()*.

11.6.1 Exemplos

Here are some examples of typical usage of the *tempfile* module:

```
>>> import tempfile

# create a temporary file and write some data to it
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# close the file, it will be removed
>>> fp.close()

# create a temporary file using a context manager
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
```

(continua na próxima página)

(continuação da página anterior)

```
# file is now closed and removed

# create a temporary directory using the context manager
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)
>>>
# directory and contents have been removed
```

11.6.2 Deprecated functions and variables

A historical way to create temporary files was to first generate a file name with the *mktemp()* function and then create a file using this name. Unfortunately this is not secure, because a different process may create a file with this name in the time between the call to *mktemp()* and the subsequent attempt to create the file by the first process. The solution is to combine the two steps and create the file immediately. This approach is used by *mkstemp()* and the other functions described above.

tempfile.**mktemp**(*suffix=''*, *prefix='tmp'*, *dir=None*)

Obsoleto desde a versão 2.3: Use *mkstemp()*.

Return an absolute pathname of a file that did not exist at the time the call is made. The *prefix*, *suffix*, and *dir* arguments are similar to those of *mkstemp()*, except that bytes file names, `suffix=None` and `prefix=None` are not supported.

Aviso: Use of this function may introduce a security hole in your program. By the time you get around to doing anything with the file name it returns, someone else may have beaten you to the punch. *mktemp()* usage can be replaced easily with *NamedTemporaryFile()*, passing it the `delete=False` parameter:

```
>>> f = NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmptjujjt'
>>> f.write(b"Hello World!\n")
13
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

11.7 :mod:glob — Expansão de padrão de nome de arquivo no estilo Unix

Código Fonte: Lib/glob.py

O módulo *glob* encontra todos os nomes de caminho que correspondem a um padrão especificado de acordo com as regras usadas pelo shell Unix, embora os resultados sejam retornados em ordem arbitrária. Nenhuma expansão de til é feita, mas `*`, `?` e os intervalos de caracteres expressos com `[]` serão correspondidos corretamente. Isso é feito usando as funções *os.scandir()* e *fnmatch.fnmatch()* em conjunto, e não invocando realmente um subshell. Observe que, ao contrário de *fnmatch.fnmatch()*, *glob* trata nomes de arquivos que começam com um ponto (`.`) como casos especiais. (Para expansão de til e variável de shell, use *os.path.expanduser()* e *os.path.expandvars()*.)

Para uma correspondência literal, coloque os metacaracteres entre colchetes. Por exemplo, `'[?]'` corresponde ao caractere `'?'`.

Ver também:

O módulo *pathlib* oferece objetos de caminho de alto nível.

`glob.glob`(*pathname*, *, *recursive=False*)
Return a possibly-empty list of path names that match *pathname*, which must be a string containing a path specification. *pathname* can be either absolute (like `/usr/src/Python-1.5/Makefile`) or relative (like `../../Tools/*/*.gif`), and can contain shell-style wildcards. Broken symlinks are included in the results (as in the shell).

Se *recursive* for verdadeiro, o padrão "`**`" corresponderá a qualquer arquivo e zero ou mais diretórios, subdiretórios e links simbólicos para diretórios. Se o padrão for seguido por um *os.sep* ou *os.altsep*, então os arquivos não irão corresponder.

Nota: Usar o padrão "`**`" em grandes árvores de diretório pode consumir uma quantidade excessiva de tempo.

Alterado na versão 3.5: Suporte a globs recursivos usando "`**`".

`glob.iglob`(*pathname*, *, *recursive=False*)
Retorna um *iterador* que produz os mesmos valores que *glob()* sem realmente armazená-los todos simultaneamente.

`glob.escape`(*pathname*)
Escapa todos os caracteres especiais (`'?'`, `'*'` e `'['`). Isso é útil se você deseja corresponder a uma string literal arbitrária que pode conter caracteres especiais. Os caracteres especiais nos pontos de compartilhamento de unidade/UNC não têm escape, por exemplo, no Windows `escape('//?/c:/Quo vadis?.txt')` retorna `'//?/c:/Quo vadis[?].txt'`.

Novo na versão 3.4.

Por exemplo, considere um diretório contendo os seguintes arquivos: `1.gif`, `2.txt`, `card.gif` e um subdiretório `sub` que contém apenas o arquivo `3.txt`. *glob()* produzirá os seguintes resultados. Observe como todos os componentes principais do caminho são preservados.

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
>>> glob.glob('**/*.txt', recursive=True)
['2.txt', 'sub/3.txt']
>>> glob.glob('./**/', recursive=True)
['./', './sub/']
```

Se o diretório contém arquivos começando com `.` eles não serão correspondidos por padrão. Por exemplo, considere um diretório contendo `card.gif` e `.card.gif`

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

Ver também:

Módulo *fnmatch* Expansão de nome de arquivo no estilo shell (não caminho)

11.8 fnmatch — Correspondência de padrões de nome de arquivo Unix

Código Fonte: Lib/fnmatch.py

Este módulo fornece suporte para curingas no estilo shell do Unix, que *não* são iguais às expressões regulares (documentadas no módulo *re*). Os caracteres especiais usados nos curingas no estilo de shell são:

Padrão	Significado
`*`	corresponde a tudo
`?`	Corresponde a qualquer caractere único
`[seq]`	corresponde a qualquer caractere em *seq*
`[!seq]`	corresponde a qualquer caractere ausente em *seq*

Para uma correspondência literal, coloque os metacaracteres entre colchetes. Por exemplo, `'[?]'` corresponde ao caractere `'?'`.

Note que o separador de nome de arquivo (`'/'` no Unix) *não* é especial para este módulo. Veja o módulo *glob* para expansão do nome do caminho (*glob* usa *filter()* para corresponder aos segmentos do nome do caminho). Da mesma forma, os nomes de arquivos que começam com um ponto final não são especiais para este módulo e são correspondidos pelos padrões `*` e `?`.

`fnmatch.`**`fnmatch`**(*filename*, *pattern*)
Testa se a string *filename* corresponde à string *pattern*, retornando *True* ou *False*. Ambos os parâmetros são normalizados em maiúsculas e minúsculas usando *os.path.normcase()*. *fnmatchcase()* pode ser usado para realizar uma comparação com distinção entre maiúsculas e minúsculas, independentemente de ser padrão para o sistema operacional.

Este exemplo vai exibir todos os nomes de arquivos no diretório atual com a extensão `.txt`:

```
import fnmatch
import os

for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print(file)
```

`fnmatch.`**`fnmatchcase`**(*filename*, *pattern*)
Testa se *filename* corresponde ao *pattern*, retornando *True* ou *False*; a comparação diferencia maiúsculas de minúsculas e não se aplica *os.path.normcase()*.

`fnmatch.`**`filter`**(*names*, *pattern*)
Retorna o subconjunto da lista de *names* que correspondem *pattern*. É o mesmo que `[n for n in names if fnmatch(n, pattern)]`, mas implementado com mais eficiência.

`fnmatch.`**`translate`**(*pattern*)
Retorna o *pattern* no estilo shell convertido em uma expressão regular para usar com *re.match()*.

Exemplo:

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'(?s:.*\\.txt)\\Z'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<re.Match object; span=(0, 10), match='foobar.txt'>
```

Ver também:

Módulo *glob* Expansão de caminho no estilo shell do Unix.

11.9 linecache — Acesso aleatório a linhas de texto

Código Fonte: Lib/linecache.py

O módulo *linecache* permite obter qualquer linha de um arquivo fonte Python, enquanto tenta otimizar internamente, usando um cache, o caso comum em que muitas linhas são lidas em um único arquivo. Isso é usado pelo módulo *traceback* para recuperar as linhas de origem para inclusão no traceback (situação da pilha de execução) formatado.

A função *tokenize.open()* é usada para abrir arquivos. Esta função usa *tokenize.detect_encoding()* para obter a codificação do arquivo; na ausência de um token de codificação, o padrão de codificação do arquivo é UTF-8.

O módulo *linecache* define as seguintes funções:

linecache.**getline**(*filename*, *lineno*, *module_globals=None*)
: Obtém a linha *lineno* do arquivo chamado *filename*. Essa função nunca levanta uma exceção — ela retornará `''` em erros (o caractere de nova linha final será incluído para as linhas encontradas).

  If a file named *filename* is not found, the function will look for it in the module search path, `sys.path`, after first checking for a **PEP 302** `__loader__` in *module_globals*, in case the module was imported from a zipfile or other non-filesystem import source.

linecache.**clearcache**()
: Limpa o cache. Use esta função se você não precisar mais de linhas de arquivos lidos anteriormente usando *getline()*.

linecache.**checkcache**(*filename=None*)
: Verifica a validade do cache. Use esta função se os arquivos no cache tiverem sido alterados no disco e você precisar da versão atualizada. Se *filename* for omitido, ele verificará todas as entradas no cache.

linecache.**lazycache**(*filename*, *module_globals*)
: Captura detalhes suficientes sobre um módulo não baseado em arquivo para permitir obter suas linhas posteriormente via *getline()* mesmo se *module_globals* for `None` na chamada posterior. Isso evita a execução de E/S até que uma linha seja realmente necessária, sem ter que carregar o módulo global indefinidamente.

  Novo na versão 3.5.

Exemplo:

```
>>> import linecache
>>> linecache.getline(linecache.__file__, 8)
'import sys\n'
```

11.10 shutil — Operações de arquivo de alto nível

Código Fonte: Lib/shutil.py

O módulo *shutil* oferece várias operações de alto nível em arquivos e coleções de arquivos. Em particular, são fornecidas funções que possuem suporte a cópia e remoção de arquivos. Para operações em arquivos individuais, veja também o módulo *os*.

Aviso: Mesmo as funções de cópia de arquivos de nível superior (*shutil.copy()*, *shutil.copy2()*) não podem copiar todos os metadados do arquivo.

Nas plataformas POSIX, isso significa que o proprietário e o grupo do arquivo são perdidos, bem como as ACLs. No Mac OS, a bifurcação de recursos e outros metadados não são usados. Isso significa que os recursos serão perdidos e o tipo de arquivo e os códigos do criador não estarão corretos. No Windows, os proprietários de arquivos, ACLs e fluxos de dados alternativos não são copiados.

11.10.1 Operações de diretório e arquivos

shutil.**copyfileobj**(*fsrc*, *fdst*[, *length*])
Copia o conteúdo do objeto do tipo arquivo *fsrc* para o objeto do tipo arquivo *fdst*. O número inteiro *length*, se fornecido, é o tamanho do buffer. Em particular, um valor negativo *length* significa copiar os dados sem repetir os dados de origem em pedaços; por padrão, os dados são lidos em pedaços para evitar o consumo descontrolado de memória. Observe que, se a posição atual do arquivo do objeto *fsrc* não for 0, apenas o conteúdo da posição atual do arquivo até o final do arquivo será copiado.

shutil.**copyfile**(*src*, *dst*, *, *follow_symlinks=True*)
Copy the contents (no metadata) of the file named *src* to a file named *dst* and return *dst*. *src* and *dst* are path names given as strings. *dst* must be the complete target file name; look at *shutil.copy()* for a copy that accepts a target directory path. If *src* and *dst* specify the same file, *SameFileError* is raised.

O local de destino deve ser gravável; caso contrário, uma exceção *OSError* será gerada. Se o *dst* já existir, ele será substituído. Arquivos especiais como dispositivos de caractere ou bloco e tubulações não podem ser copiados com esta função.

Se *follow_symlinks* for falso e *src* for um link simbólico, um novo link simbólico será criado em vez de copiar o arquivo *src* para o qual o arquivo aponta.

Alterado na versão 3.3: *IOError* costumava ser gerada em vez de *OSError*. Adicionado argumento *follow_symlinks*. Agora retorna *dst*.

Alterado na versão 3.4: Levanta *SameFileError* em vez de *Error*. Como a primeira é uma subclasse da última, essa alteração é compatível com versões anteriores.

exception shutil.**SameFileError**
Essa exceção é gerada se a origem e o destino em *copyfile()* forem o mesmo arquivo.

Novo na versão 3.4.

shutil.**copymode**(*src*, *dst*, *, *follow_symlinks=True*)
Copy the permission bits from *src* to *dst*. The file contents, owner, and group are unaffected. *src* and *dst* are path names given as strings. If *follow_symlinks* is false, and both *src* and *dst* are symbolic links, *copymode()* will attempt to modify the mode of *dst* itself (rather than the file it points to). This functionality is not available on every platform; please see *copystat()* for more information. If *copymode()* cannot modify symbolic links on the local platform, and it is asked to do so, it will do nothing and return.

Alterado na versão 3.3: Adicionado argumento *follow_symlinks*.

shutil.**copystat**(*src*, *dst*, *, *follow_symlinks=True*)

Copy the permission bits, last access time, last modification time, and flags from *src* to *dst*. On Linux, *copystat()* also copies the "extended attributes" where possible. The file contents, owner, and group are unaffected. *src* and *dst* are path names given as strings.

Se *follow_symlinks* for falso e *src* e *dst* se referirem a links simbólicos, *copystat()* operará nos próprios links simbólicos, e não nos arquivos aos quais os links simbólicos se referem - lendo as informações do link simbólico *src* e gravando as informações no link simbólico *dst*.

Nota: Nem todas as plataformas oferecem a capacidade de examinar e modificar links simbólicos. O próprio Python pode dizer qual funcionalidade está disponível localmente.

- Se `os.chmod in os.supports_follow_symlinks` for `True`, *copystat()* pode modificar os bits de permissão de um link simbólico.
- Se `os.utime in os.supports_follow_symlinks` for `True`, *copystat()* pode modificar as horas da última modificação e do último acesso de um link simbólico.
- Se `os.chflags in os.supports_follow_symlinks` for `True`, *copystat()* pode modificar os sinalizadores de um link simbólico. (`os.chflags` não está disponível em todas as plataformas.)

Nas plataformas em que algumas ou todas essas funcionalidades não estão disponíveis, quando solicitado a modificar um link simbólico, *copystat()* copiará tudo o que puder. *copystat()* nunca retorna falha.

Por favor, veja *os.supports_follow_symlinks* para mais informações.

Alterado na versão 3.3: Adicionado argumento *follow_symlinks* e suporte a atributos estendidos do Linux.

shutil.**copy**(*src*, *dst*, *, *follow_symlinks=True*)

Copia o arquivo *src* ao arquivo ou diretório *dst*. *src* e *dst* devem ser strings. Se *dst* especificar um diretório, o arquivo será copiado para *dst* usando o nome do arquivo base de *src*. Retorna o caminho para o arquivo recém-criado.

Se *follow_symlinks* for falso e *src* for um link simbólico, *dst* será criado como um link simbólico. Se *follow_symlinks* for verdadeiro e *src* for um link simbólico, *dst* será uma cópia do arquivo ao qual *src* se refere.

copy() copia os dados do arquivo e o modo de permissão do arquivo (consulte *os.chmod()*). Outros metadados, como os tempos de criação e modificação do arquivo, não são preservados. Para preservar todos os metadados do arquivo do original, use *copy2()*.

Alterado na versão 3.3: Adicionado argumento *follow_symlinks*. Agora retorna o caminho para o arquivo recém-criado.

shutil.**copy2**(*src*, *dst*, *, *follow_symlinks=True*)

Idêntico a *copy()*, exceto que *copy2()* também tenta preservar os metadados do arquivo.

When *follow_symlinks* is false, and *src* is a symbolic link, *copy2()* attempts to copy all metadata from the *src* symbolic link to the newly-created *dst* symbolic link. However, this functionality is not available on all platforms. On platforms where some or all of this functionality is unavailable, *copy2()* will preserve all the metadata it can; *copy2()* never returns failure.

copy2() usa *copystat()* para copiar os metadados do arquivo. Por favor, veja *copystat()* para obter mais informações sobre o suporte da plataforma para modificar os metadados do link simbólico.

Alterado na versão 3.3: Adicionado argumento *follow_symlinks*, tenta copiar também atributos estendidos do sistema de arquivos (atualmente apenas no Linux). Agora retorna o caminho para o arquivo recém-criado.

shutil.**ignore_patterns**(**patterns*)

Esta função de fábrica cria uma função que pode ser usada como um chamável para o argumento *ignore* de

`copytree()`, ignorando arquivos e diretórios que correspondem a um dos *patterns* de estilo glob fornecidos. Veja o exemplo abaixo.

shutil.**copytree**(*src*, *dst*, *symlinks=False*, *ignore=None*, *copy_function=copy2*, *ignore_dangling_symlinks=False*)

Recursively copy an entire directory tree rooted at *src*, returning the destination directory. The destination directory, named by *dst*, must not already exist; it will be created as well as missing parent directories. Permissions and times of directories are copied with `copystat()`, individual files are copied using `shutil.copy2()`.

Se *symlinks* for verdadeiro, os links simbólicos na árvore de origem são representados como links simbólicos na nova árvore e os metadados dos links originais serão copiados na medida do permitido pela plataforma; se falso ou omitido, o conteúdo e os metadados dos arquivos vinculados são copiados para a nova árvore.

Quando *symlinks* for falso, se o arquivo apontado pelo link simbólico não existir, uma exceção será adicionada na lista de erros gerados em uma exceção `Error` no final do processo de cópia. Você pode definir o sinalizador opcional *ignore_dangling_symlinks* como true se desejar silenciar esta exceção. Observe que esta opção não tem efeito em plataformas que não possuem suporte a `os.symlink()`.

Se *ignore* for fornecido, deve ser um chamável que receberá como argumento o diretório que está sendo visitado por `copytree()`, e uma lista de seu conteúdo, retornada por `os.listdir()`. Como `copytree()` é chamada recursivamente, o chamável *ignore* será chamado uma vez para cada diretório que é copiado. O chamável deve retornar uma sequência de nomes de diretório e arquivo em relação ao diretório atual (ou seja, um subconjunto dos itens em seu segundo argumento); esses nomes serão ignorados no processo de cópia. `ignore_patterns()` pode ser usado para criar um chamável que ignore nomes com base em padrões de estilo glob.

Se uma ou mais exceções ocorrerem, uma `Error` é levantada com uma lista dos motivos.

If *copy_function* is given, it must be a callable that will be used to copy each file. It will be called with the source path and the destination path as arguments. By default, `shutil.copy2()` is used, but any function that supports the same signature (like `shutil.copy()`) can be used.

Alterado na versão 3.3: Copia metadados quando *symlinks* for falso. Agora, retorna *dst*.

Alterado na versão 3.2: Adicionado o argumento *copy_function* para poder fornecer uma função de cópia personalizada. Adicionado o argumento *ignore_dangling_symlinks* para erros silenciosos de links simbólicos quando *symlinks* for falso.

shutil.**rmtree**(*path*, *ignore_errors=False*, *onerror=None*)

Exclui uma árvore de diretório inteira; *path* deve apontar para um diretório (mas não um link simbólico para um diretório). Se *ignore_errors* for verdadeiro, os erros resultantes de remoções com falha serão ignorados; se falso ou omitido, tais erros são tratados chamando um manipulador especificado por *onerror* ou, se for omitido, eles levantam uma exceção.

Nota: Em plataformas que suportam as funções baseadas em descritores de arquivo necessárias, uma versão resistente a ataques de links simbólicos de `rmtree()` é usada por padrão. Em outras plataformas, a implementação `rmtree()` é suscetível a um ataque de link simbólico: dados o tempo e as circunstâncias apropriados, os invasores podem manipular links simbólicos no sistema de arquivos para excluir arquivos que eles não seriam capazes de acessar de outra forma. Os aplicativos podem usar o atributo de função `rmtree.avoids_symlink_attacks` para determinar qual caso se aplica.

Se *onerror* for fornecido, deve ser um chamável que aceite três parâmetros: *function*, *path*, e *excinfo*.

O primeiro parâmetro, *function*, é a função que levantou a exceção; depende da plataforma e da implementação. O segundo parâmetro, *path*, será o nome do caminho passado para a *função*. O terceiro parâmetro, *excinfo*, será a informação de exceção retornada por `sys.exc_info()`. As exceções levantadas por *onerror* não serão detectadas.

Alterado na versão 3.3: Adicionada uma versão resistente a ataques de link simbólico que é usada automaticamente se a plataforma suportar funções baseadas em descritor de arquivo.

rmtree.**avoids_symlink_attacks**
Indica se a plataforma e implementação atuais fornecem uma versão resistente a ataques de link simbólico de *rmtree()*. Atualmente, isso só é verdade para plataformas que suportam funções de acesso ao diretório baseadas em descritor de arquivo.

Novo na versão 3.3.

shutil.**move**(*src*, *dst*, *copy_function=copy2*)
Move recursivamente um arquivo ou diretório (*src*) para outro local (*dst*) e retorna ao destino.

Se o destino for um diretório existente, *src* será movido para dentro desse diretório. Se o destino já existe, mas não é um diretório, ele pode ser sobrescrito dependendo da semântica *os.rename()*.

Se o destino está no sistema de arquivos atual, então *os.rename()* é usado. Caso contrário, *src* é copiado para *dst* usando *copy_function* e depois removido. No caso de links simbólicos, um novo link simbólico apontando para o destino de *src* será criado em ou conforme *dst* e *src* serão removidos.

If *copy_function* is given, it must be a callable that takes two arguments *src* and *dst*, and will be used to copy *src* to *dest* if *os.rename()* cannot be used. If the source is a directory, *copytree()* is called, passing it the `copy_function()`. The default *copy_function* is *copy2()*. Using *copy()* as the *copy_function* allows the move to succeed when it is not possible to also copy the metadata, at the expense of not copying any of the metadata.

Alterado na versão 3.3: Adicionada manipulação de links simbólicos explícitos para sistemas de arquivos externos, adaptando-os ao comportamento do GNU **mv**. Agora retorna *dst*.

Alterado na versão 3.5: Adicionado o argumento nomeado *copy_function*.

shutil.**disk_usage**(*path*)
Return disk usage statistics about the given path as a *named tuple* with the attributes *total*, *used* and *free*, which are the amount of total, used and free space, in bytes. On Windows, *path* must be a directory; on Unix, it can be a file or directory.

Novo na versão 3.3.

Disponibilidade: Unix, Windows.

shutil.**chown**(*path*, *user=None*, *group=None*)
Altera o proprietário *usuário* e/ou *group* do *path* fornecido.

user pode ser um nome de usuário do sistema ou um uid; o mesmo se aplica ao *group*. É necessário pelo menos um argumento.

Veja também *os.chown()*, a função subjacente.

Disponibilidade: Unix.

Novo na versão 3.3.

shutil.**which**(*cmd*, *mode=os.F_OK | os.X_OK*, *path=None*)
Retorna o caminho para um executável que seria executado se o *cmd* fornecido fosse chamado. Se nenhum *cmd* for chamado, retorna `None`.

mode é uma máscara de permissão passada para *os.access()*, por padrão determinando se o arquivo existe e é executável.

Quando nenhum *path* é especificado, os resultados de *os.environ()* são usados, retornando o valor de "PATH" ou uma alternativa de *os.defpath*.

No Windows, o diretório atual é sempre anexado ao *path*, independentemente de você usar o padrão ou fornecer o seu próprio, que é o comportamento que o shell de comando usa ao localizar executáveis. Além disso, ao encontrar o *cmd* no *path*, a variável de ambiente `PATHEXT` é verificada. Por exemplo, se você chamar `shutil.which("python")`, *which()* irá pesquisar `PATHEXT` para saber que deve procurar por `python.exe` dentro dos diretórios de *path*. Por exemplo, no Windows:

```
>>> shutil.which("python")
'C:\\Python33\\python.EXE'
```

Novo na versão 3.3.

exception shutil.**Error**

Esta exceção coleta exceções que são levantadas durante uma operação de vários arquivos. Para *copytree()*, o argumento de exceção é uma lista de tuplas de 3 elementos (*srcname*, *dstname*, *exception*).

Exemplo de copytree

Este exemplo é a implementação da função *copytree()*, descrita acima, com a docstring omitida. Ele demonstra muitas das outras funções fornecidas por este módulo.

```
def copytree(src, dst, symlinks=False):
    names = os.listdir(src)
    os.makedirs(dst)
    errors = []
    for name in names:
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks)
            else:
                copy2(srcname, dstname)
            # XXX What about devices, sockets etc.?
        except OSError as why:
            errors.append((srcname, dstname, str(why)))
        # catch the Error from the recursive copytree so that we can
        # continue with other files
        except Error as err:
            errors.extend(err.args[0])
    try:
        copystat(src, dst)
    except OSError as why:
        # can't copy file access times on Windows
        if why.winerror is None:
            errors.extend((src, dst, str(why)))
    if errors:
        raise Error(errors)
```

Outro exemplo que usa o auxiliar *ignore_patterns()*:

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

Isso irá copiar tudo, exceto os arquivos `.pyc` e arquivos ou diretórios cujo nome começa com `tmp`.

Outro exemplo que usa o argumento *ignore* para adicionar uma chamada de registro:

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []   # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```

exemplo rmtree

Este exemplo mostra como remover uma árvore de diretório no Windows onde alguns dos arquivos têm seu conjunto de bits somente leitura. Ele usa a função de retorno onerror para limpar o bit somente leitura e tentar remover novamente. Qualquer falha subsequente se propagará.

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
    func(path)

shutil.rmtree(directory, onerror=remove_readonly)
```

11.10.2 Operações de arquivamento

Novo na versão 3.2.

Alterado na versão 3.5: Adicionado suporte ao formato *xztar*.

Utilitários de alto nível para criar e ler arquivos compactados e arquivados também são fornecidos. Eles contam com os módulos `zipfile` e `tarfile`.

shutil.**make_archive**(*base_name*, *format*[, *root_dir*[, *base_dir*[, *verbose*[, *dry_run*[, *owner*[, *group*[, *logger*]]]]]]])

Cria um arquivo compactado (como zip ou tar) e retorna seu nome.

base_name é o nome do arquivo a ser criado, incluindo o caminho, menos qualquer extensão específica de formato. *format* é o formato do arquivo: um de "zip" (se o módulo `zlib` estiver disponível), "tar", "gztar" (se o módulo `zlib` estiver disponível), "bztar" (se o módulo `bz2` estiver disponível) ou "xztar" (se o módulo `lzma` estiver disponível).

root_dir is a directory that will be the root directory of the archive, all paths in the archive will be relative to it; for example, we typically chdir into *root_dir* before creating the archive.

base_dir is the directory where we start archiving from; i.e. *base_dir* will be the common prefix of all files and directories in the archive. *base_dir* must be given relative to *root_dir*. See *Archiving example with base_dir* for how to use *base_dir* and *root_dir* together.

root_dir e *base_dir* têm com padrão o diretório atual.

Se *dry_run* for verdadeiro, nenhum arquivo é criado, mas as operações que seriam executadas são registradas no *logger*.

owner e *group* são usados ao criar um arquivo tar. Por padrão, usa o proprietário e grupo atuais.

logger deve ser um objeto compatível com a **PEP 282**, geralmente uma instância de `logging.Logger`.

O argumento *verbose* não é usado e foi descontinuado.

`shutil.`**`get_archive_formats`**`()`
Retorna uma lista de formatos suportados para arquivamento. Cada elemento da sequência retornada é uma tupla `(nome, descrição)`.

Por padrão, :mod:‘shutil‘fornece estes formatos:

- *zip*: arquivo ZIP (se o módulo `zlib` estiver disponível).
- *tar*: arquivo tar não comprimido.
- *gztar*: arquivo tar compactado com gzip (se o módulo `zlib` estiver disponível).
- *bztar*: arquivo tar compactado com bzip2 (se o módulo `bz2` estiver disponível).
- *xztar*: Arquivo tar compactado com xz (se o módulo `lzma` estiver disponível).

Você pode registrar novos formatos ou fornecer seu próprio arquivador para quaisquer formatos existentes, usando `register_archive_format()`.

`shutil.`**`register_archive_format`**`(`*name*, *function*[, *extra_args*[, *description*]]`)`
Registra um arquivador para o formato *name*.

function é o chamável que será usado para descompactar arquivos. O chamável receberá o *base_name* do arquivo a ser criado, seguido pelo *base_dir* (cujo padrão é `os.curdir`) para iniciar o arquivamento. Outros argumentos são passados como argumentos nomeados *owner*, *group*, *dry_run* e *logger* (como passado em `make_archive()`).

Se fornecido, *extra_args* é uma sequência de pares `(nome, valor)` que serão usados como argumentos nomeados extras quando o arquivador chamável for usado.

description é usado por `get_archive_formats()` que retorna a lista de arquivadores. O padrão é uma string vazia.

`shutil.`**`unregister_archive_format`**`(`*name*`)`
Remove o formato de arquivo *name* da lista de formatos suportados.

`shutil.`**`unpack_archive`**`(`*filename*[, *extract_dir*[, *format*]]`)`
Descompacta um arquivo. *filename* é o caminho completo do arquivo.

extract_dir é o nome do diretório de destino onde o arquivo é descompactado. Se não for fornecido, o diretório de trabalho atual será usado.

format é o formato do arquivo: um de “zip”, “tar”, “gztar”, “bztar” ou “xztar”. Ou qualquer outro formato registrado com `register_unpack_format()`. Se não for fornecido, `unpack_archive()` irá usar a extensão do nome do arquivo e ver se um descompactador foi registrado para essa extensão. Caso nenhum seja encontrado, uma `ValueError` é levantada.

Alterado na versão 3.7: Aceita um *objeto caminho ou similar* para *filename* e *extract_dir*.

`shutil.`**`register_unpack_format`**`(`*name*, *extensions*, *function*[, *extra_args*[, *description*]]`)`
Registra um formato de descompactação. *name* é o nome do formato e *extensions* é uma lista de extensões correspondentes ao formato, como `.zip` para arquivos Zip.

function é o chamável que será usado para descompactar arquivos. O chamável receberá o caminho do arquivo, seguido pelo diretório para o qual o arquivo deve ser extraído.

Quando fornecido, *extra_args* é uma sequência de tuplas `(nome, valor)` que serão passados como argumentos nomeados para o chamável.

description pode ser fornecido para descrever o formato e será devolvido pela função `get_unpack_formats()`.

shutil.**unregister_unpack_format**(*name*)
Cancela o registro de um formato de descompactação. *name* é o nome do formato.

shutil.**get_unpack_formats**()
Retorna uma lista de todos os formatos registrados para desempacotamento. Cada elemento da sequência retornada é uma tupla `(name, extensions, description)`.

Por padrão, :mod:‘shutil‘fornece estes formatos:

- *zip*: arquivo ZIP (descompactar arquivos compactados funciona apenas se o módulo correspondente estiver disponível).
- *tar*: arquivo tar não comprimido.
- *gztar*: arquivo tar compactado com gzip (se o módulo *zlib* estiver disponível).
- *bztar*: arquivo tar compactado com bzip2 (se o módulo *bz2* estiver disponível).
- *xztar*: Arquivo tar compactado com xz (se o módulo *lzma* estiver disponível).

Você pode registrar novos formatos ou fornecer seu próprio desempacotador para quaisquer formatos existentes, usando *register_unpack_format()*.

Exemplo de arquivo

Neste exemplo, criamos um arquivo tar compactado com gzip contendo todos os arquivos encontrados no diretório `.ssh` do usuário:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

O arquivo resultante contém:

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx------ tarek/staff       0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff     609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff      65 2008-06-09 13:26:54 ./config
-rwx------ tarek/staff     668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff     609 2008-06-09 13:26:54 ./id_dsa.pub
-rw------- tarek/staff    1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff     397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff   37192 2010-02-06 18:23:10 ./known_hosts
```

Archiving example with *base_dir*

In this example, similar to the *one above*, we show how to use *make_archive()*, but this time with the usage of *base_dir*. We now have the following directory structure:

```
$ tree tmp
tmp
└── root
    └── structure
        ├── content
```

(continua na próxima página)

(continuação da página anterior)

```
            └── please_add.txt
        └── do_not_add.txt
```

In the final archive, `please_add.txt` should be included, but `do_not_add.txt` should not. Therefore we use the following:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> make_archive(
...     archive_name,
...     'tar',
...     root_dir='tmp/root',
...     base_dir='structure/content',
... )
'/Users/tarek/my_archive.tar'
```

Listing the files in the resulting archive gives us:

```
$ python -m tarfile -l /Users/tarek/myarchive.tar
structure/content/
structure/content/please_add.txt
```

11.10.3 Consultando o tamanho do terminal de saída

`shutil.`**`get_terminal_size`**(*fallback=(columns, lines)*)

Obtém o tamanho da janela do terminal.

Para cada uma das duas dimensões, a variável de ambiente, `COLUMNS` e `LINES` respectivamente, é verificada. Se a variável estiver definida e o valor for um número inteiro positivo, ela será usada.

Quando `COLUMNS` ou `LINES` não está definida, que é o caso comum, o terminal conectado a *`sys.__stdout__`* é consultado invocando *`os.get_terminal_size()`*.

Se o tamanho do terminal não pode ser consultado com sucesso, ou porque o sistema não tem suporte a consultas, ou porque não estamos conectados a um terminal, o valor dado no parâmetro `fallback` é usado. O padrão de `fallback` é `(80, 24)`, que é o tamanho padrão usado por muitos emuladores de terminal.

O valor retornado é uma tupla nomeada do tipo *`os.terminal_size`*.

Veja também: The Single UNIX Specification, Versão 2, Other Environment Variables.

Novo na versão 3.3.

11.11 `macpath` — Funções de manipulação de caminho do Mac OS 9

Código-fonte: Lib/macpath.py

Deprecated since version 3.7, will be removed in version 3.8.

Este módulo é a implementação do Mac OS 9 (e anterior) do módulo *`os.path`*. Ele pode ser usado para manipular nomes de caminhos antigos do Macintosh no Mac OS X (ou qualquer outra plataforma).

As seguintes funções estão disponíveis neste módulo: `normcase()`, `normpath()`, `isabs()`, `join()`, `split()`, `isdir()`, `isfile()`, `walk()`, `exists()`. Para outras funções disponíveis em *os.path*, contrapartes fictícias estão disponíveis.

Ver também:

Módulo *os* Interfaces do sistema operacional, incluindo funções para trabalhar com arquivos num nível inferior ao Python *file objects*.

Módulo *io* A biblioteca de E/S incorporada ao Python, incluindo as classes abstratas e algumas classes concretas, como E/S de arquivos.

Funções built-in *open()* A maneira padrão de abrir arquivos para ler e escrever em Python.

CAPÍTULO 12

Persistência de Dados

Os módulos descritos neste capítulo possuem suporte ao armazenamento de dados do Python em um formato persistente no disco. Os módulos `pickle` e `marshal` podem transformar muitos tipos de dados do Python em um fluxo de bytes e então recriar os objetos a partir dos bytes. Os vários módulos relacionados ao DBM possuem suporte a uma família de formatos de arquivo baseados em hash que armazenam um mapeamento de strings para outras strings.

A lista de módulos descritos neste capítulo é:

12.1 `pickle` — Serialização de objetos Python

Código Fonte: Lib/pickle.py

O módulo `pickle` implementa protocolos binários para serializar e desserializar uma estrutura de objeto Python. *"Pickling"* é o processo pelo qual uma hierarquia de objetos Python é convertida em um fluxo de bytes, e *"unpickling"* é a operação inversa, em que um fluxo de bytes (de um *arquivo binário* ou *objeto byte ou similar*) é convertido de volta em uma hierarquia de objetos. Pickling (e unpickling) é alternativamente conhecido como "serialização", "marshalling"[1] ou "flattening"; no entanto, para evitar confusão, os termos usados aqui são "pickling" e "unpickling".

Aviso: The `pickle` module is not secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

[1] Não confunda isso com o módulo `marshal`

12.1.1 Relacionamento com outros módulos Python

Comparação com `marshal`

Python tem um módulo de serialização mais primitivo chamado *marshal*, mas em geral *pickle* deve ser sempre a forma preferida de serializar objetos Python. *marshal* existe principalmente para oferecer suporte a arquivos `.pyc` do Python.

O módulo *pickle* difere do *marshal* de várias maneiras significativas:

- O módulo *pickle* mantém o controle dos objetos que já serializou, para que referências posteriores ao mesmo objeto não sejam serializadas novamente. *marshal* não faz isso.

  Isso tem implicações tanto para objetos recursivos quanto para compartilhamento de objetos. Objetos recursivos são objetos que contêm referências a si mesmos. Eles não são tratados pelo marshal e, de fato, tentar usar marshal em objetos recursivos irá travar seu interpretador Python. O compartilhamento de objetos ocorre quando há várias referências ao mesmo objeto em locais diferentes na hierarquia de objetos sendo serializados. *pickle* armazena tais objetos apenas uma vez, e garante que todas as outras referências apontem para a cópia mestre. Os objetos compartilhados permanecem compartilhados, o que pode ser muito importante para objetos mutáveis.

- *marshal* não pode ser usado para serializar classes definidas pelo usuário e suas instâncias. *pickle* pode salvar e restaurar instâncias de classe de forma transparente, no entanto, a definição de classe deve ser importável e viver no mesmo módulo de quando o objeto foi armazenado.

- O formato de serialização do *marshal* não tem garantia de portabilidade entre as versões do Python. Como sua principal tarefa em vida é oferecer suporte a arquivos `.pyc`, os implementadores do Python se reservam o direito de alterar o formato de serialização de maneiras não compatíveis com versões anteriores, caso haja necessidade. O formato de serialização do *pickle* tem a garantia de ser compatível com versões anteriores em todas as versões do Python, desde que um protocolo pickle compatível seja escolhido e o código de pickling e unpickling lide com diferenças de tipo Python 2 a Python 3 se seus dados estiverem cruzando aquele limite de mudança de linguagem exclusivo.

Comparação com `json`

Existem diferenças fundamentais entre os protocolos pickle e JSON (JavaScript Object Notation):

- JSON é um formato de serialização de texto (ele produz texto unicode, embora na maioria das vezes seja codificado para `utf-8`), enquanto pickle é um formato de serialização binário;
- JSON é legível por humanos, enquanto pickle não é;
- JSON é interoperável e amplamente usado fora do ecossistema Python, enquanto pickle é específico para Python;
- JSON, by default, can only represent a subset of the Python built-in types, and no custom classes; pickle can represent an extremely large number of Python types (many of them automatically, by clever usage of Python's introspection facilities; complex cases can be tackled by implementing *specific object APIs*).

Ver também:

O módulo *json*: um módulo de biblioteca padrão que permite a serialização e desserialização JSON.

12.1.2 Formato de fluxo de dados

O formato de dados usado por *pickle* é específico do Python. Isso tem a vantagem de não haver restrições impostas por padrões externos, como JSON ou XDR (que não podem representar o compartilhamento de ponteiro); no entanto, isso significa que programas não Python podem não ser capazes de reconstruir objetos Python conservados.

Por padrão, o formato de dados do *pickle* usa uma representação binária relativamente compacta. Se você precisa de características de tamanho ideal, pode com eficiência *comprimir* dados processados com pickle.

O módulo *pickletools* contém ferramentas para analisar fluxos de dados gerados por *pickle*. O código-fonte do *pickletools* tem extensos comentários sobre códigos de operações usados por protocolos de pickle.

There are currently 5 different protocols which can be used for pickling. The higher the protocol used, the more recent the version of Python needed to read the pickle produced.

- A versão 0 do protocolo é o protocolo original "legível por humanos" e é compatível com versões anteriores do Python.
- A versão 1 do protocolo é um formato binário antigo que também é compatível com versões anteriores do Python.
- A versão 2 do protocolo foi introduzida no Python 2.3. Ela fornece uma separação muito mais eficiente de *classes estilo novo*. Consulte **PEP 307** para obter informações sobre as melhorias trazidas pelo protocolo 2.
- Protocol version 3 was added in Python 3.0. It has explicit support for *bytes* objects and cannot be unpickled by Python 2.x. This is the default protocol, and the recommended protocol when compatibility with other Python 3 versions is required.
- Protocol version 4 was added in Python 3.4. It adds support for very large objects, pickling more kinds of objects, and some data format optimizations. Refer to **PEP 3154** for information about improvements brought by protocol 4.

Nota: A serialização é uma noção mais primitiva do que a persistência; embora o *pickle* leia e escreva objetos de arquivo, ele não lida com a questão de nomear objetos persistentes, nem a questão (ainda mais complicada) de acesso simultâneo a objetos persistentes. O módulo *pickle* pode transformar um objeto complexo em um fluxo de bytes e pode transformar o fluxo de bytes em um objeto com a mesma estrutura interna. Talvez a coisa mais óbvia a fazer com esses fluxos de bytes seja escrevê-los em um arquivo, mas também é concebível enviá-los através de uma rede ou armazená-los em um banco de dados. O módulo *shelve* fornece uma interface simples para fazer pickle e unpickle de objetos em arquivos de banco de dados no estilo DBM.

12.1.3 Interface do módulo

Para serializar uma hierarquia de objeto, você simplesmente chama a função *dumps()*. Da mesma forma, para desserializar um fluxo de dados, você chama a função *loads()*. No entanto, se você quiser mais controle sobre a serialização e desserialização, pode criar um objeto *Pickler* ou *Unpickler*, respectivamente.

O módulo *pickle* fornece as seguintes constantes:

pickle.**HIGHEST_PROTOCOL**
: Um inteiro, a mais alta *versão de protocolo* disponível. Este valor pode ser passado como um valor de *protocol* para as funções *dump()* e *dumps()*, bem como o construtor de *Pickler*.

pickle.**DEFAULT_PROTOCOL**
: An integer, the default *protocol version* used for pickling. May be less than *HIGHEST_PROTOCOL*. Currently the default protocol is 3, a new protocol designed for Python 3.

O módulo *pickle* fornece as seguintes funções para tornar o processo de pickling mais conveniente:

pickle.**dump**(*obj*, *file*, *protocol=None*, ***, *fix_imports=True*)
Escreve a representação após fazer pickling do objeto *obj* no objeto arquivo aberto *file*. Isso é equivalente a `Pickler(file, protocol).dump(obj)`.

O argumento opcional *protocol*, um inteiro, diz ao pickler para usar o protocolo fornecido; os protocolos suportados são de 0 a *HIGHEST_PROTOCOL*. Se não for especificado, o padrão é *DEFAULT_PROTOCOL*. Se um número negativo for especificado, *HIGHEST_PROTOCOL* é selecionado.

O argumento *file* deve ter um método write() que aceite um argumento de um único byte. Portanto, pode ser um arquivo em disco aberto para escrita binária, uma instância *io.BytesIO* ou qualquer outro objeto personalizado que atenda a esta interface.

Se *fix_imports* for verdadeiro e *protocolo* for menor que 3, pickle tentará mapear os novos nomes do Python 3 para os nomes dos módulos antigos usados no Python 2, de modo que o fluxo de dados pickle seja legível com o Python 2.

pickle.**dumps**(*obj*, *protocol=None*, ***, *fix_imports=True*)
Retorna a representação em após fazer pickling do objeto *obj* como um objeto *bytes*, ao invés de escrevê-lo em um arquivo.

Arguments *protocol* and *fix_imports* have the same meaning as in *dump()*.

pickle.**load**(*file*, ***, *fix_imports=True*, *encoding="ASCII"*, *errors="strict"*)
Lê a representação serializada com pickle de um objeto a partir de objeto arquivo aberto *file* e retorna a hierarquia de objeto reconstituído especificada nele. Isso é equivalente a `Unpickler(file).load()`.

A versão do protocolo pickle é detectada automaticamente, portanto, nenhum argumento de protocolo é necessário. Bytes após a representação serializada com pickle do objeto são ignorados.

The argument *file* must have two methods, a read() method that takes an integer argument, and a readline() method that requires no arguments. Both methods should return bytes. Thus *file* can be an on-disk file opened for binary reading, an *io.BytesIO* object, or any other custom object that meets this interface.

Optional keyword arguments are *fix_imports*, *encoding* and *errors*, which are used to control compatibility support for pickle stream generated by Python 2. If *fix_imports* is true, pickle will try to map the old Python 2 names to the new names used in Python 3. The *encoding* and *errors* tell pickle how to decode 8-bit string instances pickled by Python 2; these default to 'ASCII' and 'strict', respectively. The *encoding* can be 'bytes' to read these 8-bit string instances as bytes objects. Using `encoding='latin1'` is required for unpickling NumPy arrays and instances of *datetime*, *date* and *time* pickled by Python 2.

pickle.**loads**(*data*, ***, *fix_imports=True*, *encoding="ASCII"*, *errors="strict"*)
Retorna a hierarquia de objeto reconstituído da representação serializada com pickle *data* de um objeto. *data* deve ser um *objeto byte ou similar*.

A versão do protocolo pickle é detectada automaticamente, portanto, nenhum argumento de protocolo é necessário. Bytes após a representação serializada com pickle do objeto são ignorados.

Optional keyword arguments are *fix_imports*, *encoding* and *errors*, which are used to control compatibility support for pickle stream generated by Python 2. If *fix_imports* is true, pickle will try to map the old Python 2 names to the new names used in Python 3. The *encoding* and *errors* tell pickle how to decode 8-bit string instances pickled by Python 2; these default to 'ASCII' and 'strict', respectively. The *encoding* can be 'bytes' to read these 8-bit string instances as bytes objects. Using `encoding='latin1'` is required for unpickling NumPy arrays and instances of *datetime*, *date* and *time* pickled by Python 2.

O módulo *pickle* define três exceções:

exception pickle.**PickleError**
Classe base comum para as outras exceções de serialização com pickle. Herda *Exception*.

exception pickle.**PicklingError**
Erro levantado quando um objeto não serializável com pickle é encontrado por *Pickler*. Herda *PickleError*.

Consulte *O que pode ser serializado e desserializado com pickle?* para saber quais tipos de objetos podem ser serializados com pickle.

exception pickle.**UnpicklingError**
Erro levantado quando há um problema ao desserializar com pickle um objeto, como dados corrompidos ou violação de segurança. Herda `PickleError`.

Observe que outras exceções também podem ser levantadas durante a desserialização com pickle, incluindo (mas não necessariamente limitado a) AttributeError, EOFError, ImportError e IndexError.

The `pickle` module exports two classes, `Pickler` and `Unpickler`:

class pickle.**Pickler**(*file*, *protocol=None*, *, *fix_imports=True*)
Isso leva um arquivo binário a escrever um fluxo de dados pickle.

O argumento opcional *protocol*, um inteiro, diz ao pickler para usar o protocolo fornecido; os protocolos suportados são de 0 a `HIGHEST_PROTOCOL`. Se não for especificado, o padrão é `DEFAULT_PROTOCOL`. Se um número negativo for especificado, `HIGHEST_PROTOCOL` é selecionado.

O argumento *file* deve ter um método write() que aceite um argumento de um único byte. Portanto, pode ser um arquivo em disco aberto para escrita binária, uma instância `io.BytesIO` ou qualquer outro objeto personalizado que atenda a esta interface.

Se *fix_imports* for verdadeiro e *protocolo* for menor que 3, pickle tentará mapear os novos nomes do Python 3 para os nomes dos módulos antigos usados no Python 2, de modo que o fluxo de dados pickle seja legível com o Python 2.

dump(*obj*)
Escreve a representação serializada em pickle de *obj* no objeto arquivo aberto fornecido no construtor.

persistent_id(*obj*)
Não faz nada por padrão. Isso existe para que uma subclasse possa substituí-lo.

Se `persistent_id()` retornar `None`, *obj* é serializado com pickle como de costume. Qualquer outro valor faz com que `Pickler` emita o valor retornado como um ID persistente para *obj*. O significado deste ID persistente deve ser definido por `Unpickler.persistent_load()`. Observe que o valor retornado por `persistent_id()` não pode ter um ID persistente.

Consulte *Persistência de objetos externos* para detalhes e exemplos de usos.

dispatch_table
A tabela de despacho de um objeto pickler é um registro de *funções de redução* do tipo que pode ser declarado usando `copyreg.pickle()`. É um mapeamento cujas chaves são classes e cujos valores são funções de redução. Uma função de redução leva um único argumento da classe associada e deve estar de acordo com a mesma interface de um método `__reduce__()`.

Por padrão, um objeto pickler não terá um atributo `dispatch_table`, e em vez disso usará a tabela de despacho global gerenciada pelo módulo `copyreg`. No entanto, para personalizar a serialização com pickle de um objeto pickler específico, pode-se definir o atributo `dispatch_table` para um objeto do tipo dict. Alternativamente, se uma subclasse de `Pickler` tem um atributo `dispatch_table` então ele será usado como a tabela de despacho padrão para instâncias daquela classe.

Consulte *Tabelas de despacho* para exemplos de uso.

Novo na versão 3.3.

fast
Descontinuado. Ative o modo rápido se definido como um valor verdadeiro. O modo rápido desabilita o uso de memo, portanto, agilizando o processo de serialização com pickle por não gerar códigos de operação PUT supérfluos. Ele não deve ser usado com objetos autorreferenciais, fazer o contrário fará com que `Pickler` recorra infinitamente.

Use *pickletools.optimize()* se você precisar de serializações com pickle mais compactas.

class pickle.**Unpickler**(*file*, *, *fix_imports=True*, *encoding="ASCII"*, *errors="strict"*)

Recebe um arquivo binário para ler um fluxo de dados pickle.

A versão do protocolo do pickle é detectada automaticamente, portanto, nenhum argumento de protocolo é necessário.

The argument *file* must have two methods, a read() method that takes an integer argument, and a readline() method that requires no arguments. Both methods should return bytes. Thus *file* can be an on-disk file object opened for binary reading, an *io.BytesIO* object, or any other custom object that meets this interface.

Optional keyword arguments are *fix_imports*, *encoding* and *errors*, which are used to control compatibility support for pickle stream generated by Python 2. If *fix_imports* is true, pickle will try to map the old Python 2 names to the new names used in Python 3. The *encoding* and *errors* tell pickle how to decode 8-bit string instances pickled by Python 2; these default to 'ASCII' and 'strict', respectively. The *encoding* can be 'bytes' to read these 8-bit string instances as bytes objects.

load()

Lê a representação serializada com pickle de um objeto a partir do objeto arquivo aberto fornecido no construtor e retorna a hierarquia de objeto reconstituído especificada nele. Os bytes após a representação serializada com pickle do objeto são ignorados.

persistent_load(*pid*)

Levanta um *UnpicklingError* por padrão.

Se definido, *persistent_load()* deve retornar o objeto especificado pelo ID persistente *pid*. Se um ID persistente inválido for encontrado, uma *UnpicklingError* deve ser levantada.

Consulte *Persistência de objetos externos* para detalhes e exemplos de usos.

find_class(*module*, *name*)

Importa *module* se necessário e retorna o objeto chamado *name* dele, onde os argumentos *module* e *name* são objetos **:classe:'str'**. Observe, ao contrário do que seu nome sugere, *find_class()* também é usado para encontrar funções.

As subclasses podem substituir isso para obter controle sobre quais tipos de objetos e como eles podem ser carregados, reduzindo potencialmente os riscos de segurança. Confira *Restringindo globais* para detalhes.

12.1.4 O que pode ser serializado e desserializado com pickle?

Os seguintes tipos podem ser serializados com pickle:

- `None`, `True`, e `False`
- inteiros, números de ponto flutuante, números complexos
- strings, bytes, bytearrays
- tuplas, listas, conjuntos e dicionários contendo apenas objetos serializáveis com pickle
- funções definidas no nível superior de um módulo (usando `def`, não `lambda`)
- funções embutidas definidas no nível superior de um módulo
- classes que são definidas no nível superior de um módulo
- instâncias de classes cujo *__dict__* ou o resultado da chamada de *__getstate__()* seja serializável com pickle (veja a seção *Serializando com pickle instâncias de classes* para detalhes).

As tentativas de serializar objetos não serializáveis com pickle vão levantar a exceção *PicklingError*; quando isso acontece, um número não especificado de bytes pode já ter sido escrito no arquivo subjacente. Tentar serializar com pickle

uma estrutura de dados altamente recursiva pode exceder a profundidade máxima de recursão, a *RecursionError* será levantada neste caso. Você pode aumentar este limite cuidadosamente com *sys.setrecursionlimit()*.

Observe que as funções (embutidas e definidas pelo usuário) são serializadas com pickle por referência de nome "totalmente qualificado", não por valor.[2] Isso significa que apenas o nome da função é serializado com pickle, junto com o nome do módulo no qual a função está definida. Nem o código da função, nem qualquer um de seus atributos de função são serializados com pickle. Assim, o módulo de definição deve ser importável no ambiente de desserialização com pickle, e o módulo deve conter o objeto nomeado, caso contrário, uma exceção será levantada.[3]

Da mesma forma, as classes são serializadas com pickle por referência nomeada, portanto, aplicam-se as mesmas restrições no ambiente de desserialização com pickle. Observe que nenhum código ou dado da classe é coletado, portanto, no exemplo a seguir, o atributo de classe `attr` não é restaurado no ambiente de desserialização com pickle:

```
class Foo:
    attr = 'A class attribute'

picklestring = pickle.dumps(Foo)
```

Essas restrições são a razão pela qual as funções e classes serializáveis com pickle devem ser definidas no nível superior de um módulo.

Da mesma forma, quando as instâncias da classe são serializadas com pickle, o código e os dados de sua classe não são serializados junto com elas. Apenas os dados da instância são serializados com pickle. Isso é feito propositalmente, para que você possa corrigir bugs em uma classe ou adicionar métodos à classe e ainda carregar objetos que foram criados com uma versão anterior da classe. Se você planeja ter objetos de longa duração que verão muitas versões de uma classe, pode valer a pena colocar um número de versão nos objetos para que as conversões adequadas possam ser feitas pelo método *__setstate__()* da classe.

12.1.5 Serializando com pickle instâncias de classes

Nesta seção, descrevemos os mecanismos gerais disponíveis para você definir, personalizar e controlar como as instâncias de classe são serializadas e desserializadas com pickle.

Na maioria dos casos, nenhum código adicional é necessário para tornar as instâncias serializáveis com pickle. Por padrão, o pickle recuperará a classe e os atributos de uma instância por meio de introspecção. Quando uma instância de classe não está serializada com pickle, seu método `__init__()` geralmente *não* é invocado. O comportamento padrão primeiro cria uma instância não inicializada e, em seguida, restaura os atributos salvos. O código a seguir mostra uma implementação desse comportamento:

```
def save(obj):
    return (obj.__class__, obj.__dict__)

def load(cls, attributes):
    obj = cls.__new__(cls)
    obj.__dict__.update(attributes)
    return obj
```

As classes podem alterar o comportamento padrão, fornecendo um ou vários métodos especiais:

`object.`**`__getnewargs_ex__`**`()`

Nos protocolos 2 e mais recentes, as classes que implementam o método *__getnewargs_ex__()* podem ditar os valores passados para o método `__new__()` após a desserialização com pickle. O método deve retornar um par `(args, kwargs)` onde *args* é uma tupla de argumentos posicionais e *kwargs* um dicionário de argumentos nomeados para construir o objeto. Esses serão passados para o método `__new__()` após a desserialização com pickle.

[2] É por isso que funções `lambda` não podem ser serializadas com pickle: todas as funções `lambda` compartilham o mesmo nome: `<lambda>`.
[3] A exceção levantada provavelmente será uma *ImportError* ou uma *AttributeError*, mas poderia ser outra coisa.

Você deve implementar este método se o método `__new__()` de sua classe requer argumentos somente-nomeados. Caso contrário, é recomendado para compatibilidade implementar *__getnewargs__()*.

Alterado na versão 3.6: *__getnewargs_ex__()* é agora usado em protocolos 2 e 3.

`object.`**`__getnewargs__`**`()`
Este método serve a um propósito semelhante ao de *__getnewargs_ex__()*, mas tem suporte apenas a argumentos posicionais. Ele deve retornar uma tupla de argumentos `args` que serão passados para o método `__new__()` após a desserialização com pickle.

__getnewargs__() não será chamado se *__getnewargs_ex__()* estiver definido.

Alterado na versão 3.6: Antes do Python 3.6, *__getnewargs__()* era chamado em vez de *__getnewargs_ex__()* nos protocolos 2 e 3.

`object.`**`__getstate__`**`()`
As classes podem influenciar ainda mais como suas instâncias são serializadas com pickle; se a classe define o método *__getstate__()*, ele é chamado e o objeto retornado é serializado com pickle como o conteúdo da instância, ao invés do conteúdo do dicionário da instância. Se o método *__getstate__()* estiver ausente, o *__dict__* da instância é serializado com pickle como de costume.

`object.`**`__setstate__`**`(`*state*`)`
Ao desserializar com pickle, se a classe define *__setstate__()*, ela é chamada com o estado não desserializado. Nesse caso, não há nenhum requisito para que o objeto de estado seja um dicionário. Caso contrário, o estado serializado com pickle deve ser um dicionário e seus itens são atribuídos ao dicionário da nova instância.

Nota: Se *__getstate__()* retornar um valor falso, o método *__setstate__()* não será chamado quando da desserialização com pickle.

Confira a seção *Manipulação de objetos com estado* para mais informações sobre como usar os métodos *__getstate__()* e *__setstate__()*.

Nota: Quando da desserialização com pickle, alguns métodos, como `__getattr__()`, `__getattribute__()` ou `__setattr__()`, podem ser chamados na instância. No caso desses métodos dependerem de alguma invariante interna ser verdadeira, o tipo deve ser implementado `__new__()` para estabelecer tal invariante, pois `__init__()` não é chamada quando da desserialização com pickle em uma instância.

Como veremos, o pickle não usa diretamente os métodos descritos acima. Na verdade, esses métodos são parte do protocolo de cópia que implementa o método especial *__reduce__()*. O protocolo de cópia fornece uma interface unificada para recuperar os dados necessários para serialização com pickle e cópia de objetos.[4]

Apesar de poderoso, implementar *__reduce__()* diretamente em sua classe é algo propenso a erro. Por este motivo, designers de classe devem usar a interface de alto nível (ou seja, *__getnewargs_ex__()*, *__getstate__()* e *__setstate__()*) sempre que possível. Vamos mostrar, porém, casos em que o uso de *__reduce__()* é a única opção ou leva a uma serialização com pickle mais eficiente, ou as ambas.

`object.`**`__reduce__`**`()`
A interface está atualmente definida da seguinte maneira. O método *__reduce__()* não aceita nenhum argumento e deve retornar uma string ou preferencialmente uma tupla (o objeto retornado é frequentemente referido como o "valor de redução").

Se uma string é retornada, ela deve ser interpretada como o nome de uma variável global. Deve ser o nome local do objeto relativo ao seu módulo; o módulo pickle pesquisa o espaço de nomes do módulo para determinar o módulo do objeto. Esse comportamento é normalmente útil para singletons.

[4] O módulo *copy* usa este protocolo para operações de cópia rasa e cópia profunda.

When a tuple is returned, it must be between two and five items long. Optional items can either be omitted, or `None` can be provided as their value. The semantics of each item are in order:

- Um objeto chamável que será chamado para criar a versão inicial do objeto.
- Uma tupla de argumentos para o objeto chamável. Uma tupla vazia deve ser fornecida se o chamável não aceitar nenhum argumento.
- Opcionalmente, o estado do objeto, que será passado para o método *__setstate__()* do objeto conforme descrito anteriormente. Se o objeto não tiver tal método, o valor deve ser um dicionário e será adicionado ao atributo *__dict__* do objeto.
- Opcionalmente, um iterador (e não uma sequência) produzindo itens sucessivos. Esses itens serão anexados ao objeto usando `obj.append(item)` ou, em lote, usando `obj.extend(lista_de_itens)`. Isso é usado principalmente para subclasses de lista, mas pode ser usado por outras classes, desde que tenham os métodos `append()` e `extend()` com a assinatura apropriada. (Se `append()` ou `extend()` é usado depende de qual versão do protocolo pickle é usada, bem como o número de itens a anexar, então ambos devem ser suportados.)
- Opcionalmente, um iterador (não uma sequência) produzindo pares de valor-chave sucessivos. Esses itens serão armazenados no objeto usando `obj[chave]=valor`. Isso é usado principalmente para subclasses de dicionário, mas pode ser usado por outras classes, desde que implementem `__setitem__()`.

`object.`**`__reduce_ex__`**(*protocol*)

Alternativamente, um método *__reduce_ex__()* pode ser definido. A única diferença é que este método deve ter um único argumento inteiro, a versão do protocolo. Quando definido, pickle irá preferir isso ao método *__reduce__()*. Além disso, *__reduce__()* automaticamente se torna um sinônimo para a versão estendida. O principal uso desse método é fornecer valores de redução com compatibilidade reversa para versões mais antigas do Python.

Persistência de objetos externos

Para o benefício da persistência do objeto, o módulo *pickle* tem suporte à noção de uma referência a um objeto fora do fluxo de dados serializados com pickle. Esses objetos são referenciados por um ID persistente, que deve ser uma string de caracteres alfanuméricos (para o protocolo 0)[5] ou apenas um objeto arbitrário (para qualquer protocolo mais recente).

A resolução de tais IDs persistentes não é definida pelo módulo *pickle*; ele vai delegar esta resolução aos métodos definidos pelo usuário no selecionador e no separador, *persistent_id()* e *persistent_load()* respectivamente.

Para serializar com pickle objetos que têm um ID externo persistente, o pickler deve ter um método *persistent_id()* personalizado que recebe um objeto como um argumento e retorna `None` ou o ID persistente para esse objeto. Quando `None` é retornado, o pickler simplesmente serializa o objeto normalmente. Quando uma string de ID persistente é retornada, o pickler serializa aquele objeto, junto com um marcador para que o unpickler o reconheça como um ID persistente.

Para desserializar com pickle objetos externos, o unpickler deve ter um método *persistent_load()* personalizado que recebe um objeto de ID persistente e retorna o objeto referenciado.

Aqui está um exemplo abrangente que apresenta como o ID persistente pode ser usado para serializar com pickle objetos externos por referência.

```
# Simple example presenting how persistent ID can be used to pickle
# external objects by reference.

import pickle
import sqlite3
```

(continua na próxima página)

[5] A limitação de caracteres alfanuméricos se deve ao fato dos IDs persistentes, no protocolo 0, serem delimitados pelo caractere de nova linha. Portanto, se qualquer tipo de caractere de nova linha ocorrer em IDs persistentes, o pickle resultante se tornará ilegível.

(continuação da página anterior)

```
from collections import namedtuple

# Simple class representing a record in our database.
MemoRecord = namedtuple("MemoRecord", "key, task")

class DBPickler(pickle.Pickler):

    def persistent_id(self, obj):
        # Instead of pickling MemoRecord as a regular class instance, we emit a
        # persistent ID.
        if isinstance(obj, MemoRecord):
            # Here, our persistent ID is simply a tuple, containing a tag and a
            # key, which refers to a specific record in the database.
            return ("MemoRecord", obj.key)
        else:
            # If obj does not have a persistent ID, return None. This means obj
            # needs to be pickled as usual.
            return None


class DBUnpickler(pickle.Unpickler):

    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection

    def persistent_load(self, pid):
        # This method is invoked whenever a persistent ID is encountered.
        # Here, pid is the tuple returned by DBPickler.
        cursor = self.connection.cursor()
        type_tag, key_id = pid
        if type_tag == "MemoRecord":
            # Fetch the referenced record from the database and return it.
            cursor.execute("SELECT * FROM memos WHERE key=?", (str(key_id),))
            key, task = cursor.fetchone()
            return MemoRecord(key, task)
        else:
            # Always raises an error if you cannot return the correct object.
            # Otherwise, the unpickler will think None is the object referenced
            # by the persistent ID.
            raise pickle.UnpicklingError("unsupported persistent object")


def main():
    import io
    import pprint

    # Initialize and populate our database.
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE memos(key INTEGER PRIMARY KEY, task TEXT)")
    tasks = (
        'give food to fish',
        'prepare group meeting',
        'fight with a zebra',
        )
    for task in tasks:
```

(continua na próxima página)

(continuação da página anterior)

```
        cursor.execute("INSERT INTO memos VALUES(NULL, ?)", (task,))

    # Fetch the records to be pickled.
    cursor.execute("SELECT * FROM memos")
    memos = [MemoRecord(key, task) for key, task in cursor]
    # Save the records using our custom DBPickler.
    file = io.BytesIO()
    DBPickler(file).dump(memos)

    print("Pickled records:")
    pprint.pprint(memos)

    # Update a record, just for good measure.
    cursor.execute("UPDATE memos SET task='learn italian' WHERE key=1")

    # Load the records from the pickle data stream.
    file.seek(0)
    memos = DBUnpickler(file, conn).load()

    print("Unpickled records:")
    pprint.pprint(memos)


if __name__ == '__main__':
    main()
```

Tabelas de despacho

Se alguém quiser personalizar a serialização com pickle de algumas classes sem perturbar nenhum outro código que dependa da serialização, pode-se criar um pickler com uma tabela de despacho privada.

A tabela de despacho global gerenciada pelo módulo *copyreg* está disponível como `copyreg.dispatch_table`. Portanto, pode-se escolher usar uma cópia modificada de `copyreg.dispatch_table` como uma tabela de despacho privada.

Por exemplo

```
f = io.BytesIO()
p = pickle.Pickler(f)
p.dispatch_table = copyreg.dispatch_table.copy()
p.dispatch_table[SomeClass] = reduce_SomeClass
```

cria uma instância de *pickle.Pickler* com uma tabela de despacho privada que trata a classe `SomeClass` especialmente. Alternativamente, o código

```
class MyPickler(pickle.Pickler):
    dispatch_table = copyreg.dispatch_table.copy()
    dispatch_table[SomeClass] = reduce_SomeClass
f = io.BytesIO()
p = MyPickler(f)
```

faz o mesmo, mas todas as instâncias de `MyPickler` irão por padrão compartilhar a mesma tabela de despacho. O código equivalente usando o módulo *copyreg* é

```
copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)
```

Manipulação de objetos com estado

Aqui está um exemplo que mostra como modificar o comportamento de serialização com pickle de uma classe. A classe `TextReader` abre um arquivo texto e retorna o número da linha e o conteúdo da linha cada vez que seu método `readline()` é chamado. Se uma instância de `TextReader` for selecionada, todos os atributos *exceto* o membro do objeto arquivo são salvos. Quando a instância é removida, o arquivo é reaberto e a leitura continua a partir do último local. Os métodos *__setstate__()* e *__getstate__()* são usados para implementar este comportamento.

```
class TextReader:
    """Print and number lines in a text file."""

    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename)
        self.lineno = 0

    def readline(self):
        self.lineno += 1
        line = self.file.readline()
        if not line:
            return None
        if line.endswith('\n'):
            line = line[:-1]
        return "%i: %s" % (self.lineno, line)

    def __getstate__(self):
        # Copy the object's state from self.__dict__ which contains
        # all our instance attributes. Always use the dict.copy()
        # method to avoid modifying the original state.
        state = self.__dict__.copy()
        # Remove the unpicklable entries.
        del state['file']
        return state

    def __setstate__(self, state):
        # Restore instance attributes (i.e., filename and lineno).
        self.__dict__.update(state)
        # Restore the previously opened file's state. To do so, we need to
        # reopen it and read from it until the line count is restored.
        file = open(self.filename)
        for _ in range(self.lineno):
            file.readline()
        # Finally, save the file.
        self.file = file
```

Um exemplo de uso pode ser algo assim:

```
>>> reader = TextReader("hello.txt")
>>> reader.readline()
'1: Hello world!'
>>> reader.readline()
```

(continua na próxima página)

(continuação da página anterior)

```
'2: I am line number two.'
>>> new_reader = pickle.loads(pickle.dumps(reader))
>>> new_reader.readline()
'3: Goodbye!'
```

12.1.6 Restringindo globais

Por padrão, a desserialização com pickle importará qualquer classe ou função que encontrar nos dados pickle. Para muitos aplicativos, esse comportamento é inaceitável, pois permite que o unpickler importe e invoque código arbitrário. Basta considerar o que este fluxo de dados pickle feito à mão faz quando carregado:

```
>>> import pickle
>>> pickle.loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
hello world
0
```

Neste exemplo, o unpickler importa a função *os.system()* e então aplica o argumento string "echo hello world". Embora este exemplo seja inofensivo, não é difícil imaginar um que possa danificar seu sistema.

Por esta razão, você pode querer controlar o que é desserializado com pickle personalizando *Unpickler.find_class()*. Ao contrário do que seu nome sugere, *Unpickler.find_class()* é chamado sempre que um global (ou seja, uma classe ou uma função) é solicitado. Assim, é possível proibir completamente os globais ou restringi-los a um subconjunto seguro.

Aqui está um exemplo de um unpickler que permite que apenas algumas classes seguras do módulo *builtins* sejam carregadas:

```
import builtins
import io
import pickle

safe_builtins = {
    'range',
    'complex',
    'set',
    'frozenset',
    'slice',
}

class RestrictedUnpickler(pickle.Unpickler):

    def find_class(self, module, name):
        # Only allow safe classes from builtins.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # Forbid everything else.
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                     (module, name))

def restricted_loads(s):
    """Helper function analogous to pickle.loads()."""
    return RestrictedUnpickler(io.BytesIO(s)).load()
```

Um exemplo de uso do nosso unpickler funcionando como esperado:

```
>>> restricted_loads(pickle.dumps([1, 2, range(15)]))
[1, 2, range(0, 15)]
>>> restricted_loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'os.system' is forbidden
>>> restricted_loads(b'cbuiltins\neval\n'
...                  b'(S\'getattr(__import__("os"), "system")'
...                  b'("echo hello world")\'\ntR.')
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'builtins.eval' is forbidden
```

Como nossos exemplos mostram, você deve ter cuidado com o que permite que seja desserializado com pickle. Portanto, se a segurança é uma preocupação, você pode querer considerar alternativas como a API de marshalling em *xmlrpc.client* ou soluções de terceiros.

12.1.7 Performance

Versões recentes do protocolo pickle (do protocolo 2 em diante) apresentam codificações binárias eficientes para vários recursos comuns e tipos embutidos. Além disso, o módulo *pickle* tem um otimizador transparente escrito em C.

12.1.8 Exemplos

Para código mais simples, use as funções *dump()* e *load()*.

```
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

O exemplo a seguir lê os dados resultantes em serializados com pickle.

```
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
```

Ver também:

Módulo *copyreg* Registro de construtor de interface Pickle para tipos de extensão.

Módulo *pickletools* Ferramentas para trabalhar e analisar dados serializados com pickle.

Módulo *shelve* Banco de dados indexado de objetos; usa *pickle*.

Módulo *copy* Cópia rasa e cópia profunda de objeto.

Módulo *marshal* Serialização de alto desempenho de tipos embutidos.

12.2 copyreg — Registra funções de suporte pickle

Código-fonte: Lib/copyreg.py

O módulo *copyreg* oferece uma maneira de definir as funções usadas durante a remoção de objetos específicos. Os módulos *pickle* e *copy* usam essas funções ao selecionar/copiar esses objetos. O módulo fornece informações de configuração sobre construtores de objetos que não são classes. Esses construtores podem ser funções de fábrica ou instâncias de classes.

copyreg.**constructor**(*object*)
Declara *object* para ser um construtor válido. Se *object* não puder ser chamado (e, portanto, não for válido como um construtor), gera *TypeError*.

copyreg.**pickle**(*type*, *function*, *constructor=None*)
Declara que *function* deve ser usada como uma função de "redução" para objetos do tipo *type*. *function* deve retornar uma string ou uma tupla contendo dois ou três elementos.

O parâmetro opcional *constructor*, se fornecido, é um objeto que pode ser chamado, que pode ser usado para reconstruir o objeto quando chamado com a tupla de argumentos retornados por *function* no tempo de decapagem. *TypeError* será gerado se *object* for uma classe ou *constructor* não for chamado.

Veja o módulo *pickle* para mais detalhes sobre a interface esperada de *function* e *constructor*. Note que o atributo *dispatch_table* de um objeto pickler ou subclasse de *pickle.Pickler* também podem ser usados para declarar funções de redução.

12.2.1 Exemplo

O exemplo abaixo gostaria de mostrar como registrar uma função de pickle e como ela será usada:

```
>>> import copyreg, copy, pickle
>>> class C(object):
...     def __init__(self, a):
...         self.a = a
...
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
...
>>> copyreg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```

12.3 shelve — Persistência de objetos Python

Código Fonte: Lib/shelve.py

A "shelf" is a persistent, dictionary-like object. The difference with "dbm" databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects — anything that the *pickle* module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys are ordinary strings.

shelve.**open**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

Open a persistent dictionary. The filename specified is the base filename for the underlying database. As a side-effect, an extension may be added to the filename and more than one file may be created. By default, the underlying database file is opened for reading and writing. The optional *flag* parameter has the same interpretation as the *flag* parameter of *dbm.open()*.

By default, version 3 pickles are used to serialize values. The version of the pickle protocol can be specified with the *protocol* parameter.

Because of Python semantics, a shelf cannot know when a mutable persistent-dictionary entry is modified. By default modified objects are written *only* when assigned to the shelf (see *Exemplo*). If the optional *writeback* parameter is set to `True`, all entries accessed are also cached in memory, and written back on *sync()* and *close()*; this can make it handier to mutate mutable entries in the persistent dictionary, but, if many entries are accessed, it can consume vast amounts of memory for the cache, and it can make the close operation very slow since all accessed entries are written back (there is no way to determine which accessed entries are mutable, nor which ones were actually mutated).

Nota: Do not rely on the shelf being closed automatically; always call *close()* explicitly when you don't need it any more, or use *shelve.open()* as a context manager:

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

Aviso: Because the *shelve* module is backed by *pickle*, it is insecure to load a shelf from an untrusted source. Like with pickle, loading a shelf can execute arbitrary code.

Shelf objects support all methods supported by dictionaries. This eases the transition from dictionary based scripts to those requiring persistent storage.

Two additional methods are supported:

Shelf.**sync**()

Write back all entries in the cache if the shelf was opened with *writeback* set to *True*. Also empty the cache and synchronize the persistent dictionary on disk, if feasible. This is called automatically when the shelf is closed with *close()*.

Shelf.**close**()

Synchronize and close the persistent *dict* object. Operations on a closed shelf will fail with a *ValueError*.

Ver também:

Persistent dictionary recipe with widely supported storage formats and having the speed of native dictionaries.

12.3.1 Restrições

- The choice of which database package will be used (such as *dbm.ndbm* or *dbm.gnu*) depends on which interface is available. Therefore it is not safe to open the database directly using *dbm*. The database is also (unfortunately) subject to the limitations of *dbm*, if it is used — this means that (the pickled representation of) the objects stored in the database should be fairly small, and in rare cases key collisions may cause the database to refuse updates.
- The *shelve* module does not support *concurrent* read/write access to shelved objects. (Multiple simultaneous read accesses are safe.) When a program has a shelf open for writing, no other program should have it open for reading or writing. Unix file locking can be used to solve this, but this differs across Unix versions and requires knowledge about the database implementation used.

class shelve.**Shelf**(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)
A subclass of *collections.abc.MutableMapping* which stores pickled values in the *dict* object.

By default, version 3 pickles are used to serialize values. The version of the pickle protocol can be specified with the *protocol* parameter. See the *pickle* documentation for a discussion of the pickle protocols.

If the *writeback* parameter is `True`, the object will hold a cache of all entries accessed and write them back to the *dict* at sync and close times. This allows natural operations on mutable entries, but can consume much more memory and make sync and close take a long time.

The *keyencoding* parameter is the encoding used to encode keys before they are used with the underlying dict.

A *Shelf* object can also be used as a context manager, in which case it will be automatically closed when the `with` block ends.

Alterado na versão 3.2: Added the *keyencoding* parameter; previously, keys were always encoded in UTF-8.

Alterado na versão 3.4: Adicionado suporte a gerenciador de contexto.

class shelve.**BsdDbShelf**(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)
A subclass of *Shelf* which exposes `first()`, `next()`, `previous()`, `last()` and `set_location()` which are available in the third-party `bsddb` module from pybsddb but not in other database modules. The *dict* object passed to the constructor must support those methods. This is generally accomplished by calling one of `bsddb.hashopen()`, `bsddb.btopen()` or `bsddb.rnopen()`. The optional *protocol*, *writeback*, and *keyencoding* parameters have the same interpretation as for the *Shelf* class.

class shelve.**DbfilenameShelf**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)
A subclass of *Shelf* which accepts a *filename* instead of a dict-like object. The underlying file will be opened using *dbm.open()*. By default, the file will be created and opened for both read and write. The optional *flag* parameter has the same interpretation as for the *open()* function. The optional *protocol* and *writeback* parameters have the same interpretation as for the *Shelf* class.

12.3.2 Exemplo

To summarize the interface (`key` is a string, `data` is an arbitrary object):

```
import shelve

d = shelve.open(filename)  # open -- file may get suffix added by low-level
                           # library

d[key] = data              # store data at key (overwrites old data if
                           # using an existing key)
data = d[key]              # retrieve a COPY of data at key (raise KeyError
                           # if no such key)
```

(continua na próxima página)

(continuação da página anterior)

```
del d[key]                  # delete data stored at key (raises KeyError
                            # if no such key)

flag = key in d             # true if the key exists
klist = list(d.keys())      # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]         # this works as expected, but...
d['xx'].append(3)           # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']              # extracts the copy
temp.append(5)              # mutates the copy
d['xx'] = temp              # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()                   # close it
```

Ver também:

Módulo *dbm* Generic interface to dbm-style databases.

Módulo *pickle* Object serialization used by *shelve*.

12.4 marshal — Serialização interna de objetos Python

Este módulo contém funções que podem ler e gravar valores Python em formato binário. O formato é específico para Python, mas independente dos problemas de arquitetura da máquina (por exemplo, você pode gravar um valor Python em um arquivo em um PC, transportar o arquivo para um Sun e lê-lo de volta lá). Os detalhes do formato não são documentados propositalmente; ele pode mudar entre as versões do Python (embora raramente mude).[1]

Este não é um módulo de "persistência" geral. Para persistência geral e transferência de objetos Python através de chamadas RPC, veja os módulos *pickle* e *shelve*. O módulo *marshal* existe principalmente para ter suporte à leitura e escrita do código "pseudocompilado" para módulos Python de arquivos `.pyc`. Portanto, os mantenedores do Python se reservam o direito de modificar o formato do marshal de maneiras incompatíveis com versões anteriores, caso seja necessário. Se você estiver serializando e desserializando objetos Python, use o módulo *pickle* – o desempenho é comparável, a independência de versão é garantida e pickle tem suporte a uma gama substancialmente maior de objetos do que marshal.

> **Aviso:** O módulo *marshal* não se destina a ser seguro contra dados errôneos ou construídos de forma maliciosa. Nunca faça o unmarshalling de dados recebidos de uma fonte não confiável ou não autenticada.

Nem todos os tipos de objetos Python são suportados; em geral, apenas objetos cujo valor é independente de uma invocação particular de Python podem ser escritos e lidos por este módulo. Os seguintes tipos são suportados: booleanos,

[1] O nome deste módulo deriva de um pouco da terminologia usada pelos designers do Modula-3 (entre outros), que usam o termo "marshalling" para enviar dados em um formato independente. Estritamente falando, "to marshal" significa converter alguns dados da forma interna para a externa (em um buffer RPC, por exemplo) e "unmarshalling" para o processo reverso.

inteiros, números de ponto flutuante, números complexos, strings, bytes, bytearrays, tuplas, listas, conjuntos, frozensets, dicionários e objetos código, onde deve ser entendido que tuplas, listas, conjuntos, frozensets e os dicionários são suportados apenas enquanto os próprios valores contidos neles forem suportados. Os singletons `None`, `Ellipsis` e `StopIteration` também podem ser serializados e dessarializados com marshal. Para formato *version* inferior a 3, listas recursivas, conjuntos e dicionários não podem ser escritos (veja abaixo).

Existem funções que leem/gravam arquivos, bem como funções que operam em objetos byte ou similares.

O módulo define estas funções:

`marshal.`**`dump`**(*value*, *file*[, *version*])
Grava o valor no arquivo aberto. O valor deve ser um tipo compatível. O arquivo deve ser *arquivo binário* gravável.

Se o valor tem (ou contém um objeto que tem) um tipo não suportado, uma exceção `ValueError` é levantada – mas dados de lixo também serão gravados no arquivo. O objeto não será lido corretamente por `load()`.

O argumento *version* indica o formato de dados que o `dump` deve usar (veja abaixo).

`marshal.`**`load`**(*file*)
Lê um valor do arquivo aberto e retorna-o. Se nenhum valor válido for lido (por exemplo, porque os dados têm um formato de empacotamento incompatível com uma versão diferente do Python), levanta `EOFError`, `ValueError` ou `TypeError`. O arquivo deve ser um *arquivo binário* legível.

Nota: Se um objeto contendo um tipo não suportado foi empacotado com `dump()`, `load()` irá substituir `None` pelo tipo não empacotável.

`marshal.`**`dumps`**(*value*[, *version*])
Retorna o objeto bytes que seria escrito em um arquivo por `dump(value, file)`. O valor deve ser um tipo compatível. Levanta uma exceção `ValueError` se o valor tem (ou contém um objeto que tem) um tipo não suportado.

O argumento *version* indica o formato de dados que `dumps` deve usar (veja abaixo).

`marshal.`**`loads`**(*bytes*)
Converte o *objeto byte ou similar* em um valor. Se nenhum valor válido for encontrado, levanta `EOFError`, `ValueError` ou `TypeError`. Bytes extras na entrada são ignorados.

Além disso, as seguintes constantes são definidas:

`marshal.`**`version`**
Indica o formato que o módulo usa. A versão 0 é o formato histórico, a versão 1 compartilha strings internas e a versão 2 usa um formato binário para números de ponto flutuante. A versão 3 adiciona suporte para instanciação e recursão de objetos. A versão atual é 4.

12.5 `dbm` — Interfaces to Unix "databases"

Código Fonte: Lib/dbm/__init__.py

`dbm` is a generic interface to variants of the DBM database — `dbm.gnu` or `dbm.ndbm`. If none of these modules is installed, the slow-but-simple implementation in module `dbm.dumb` will be used. There is a third party interface to the Oracle Berkeley DB.

`exception` dbm.**`error`**
A tuple containing the exceptions that can be raised by each of the supported modules, with a unique exception also named `dbm.error` as the first item — the latter is used when `dbm.error` is raised.

dbm.**whichdb**(*filename*)
This function attempts to guess which of the several simple database modules available — *dbm.gnu*, *dbm.ndbm* or *dbm.dumb* — should be used to open a given file.

Returns one of the following values: `None` if the file can't be opened because it's unreadable or doesn't exist; the empty string (`''`) if the file's format can't be guessed; or a string containing the required module name, such as `'dbm.ndbm'` or `'dbm.gnu'`.

dbm.**open**(*file*, *flag='r'*, *mode=0o666*)
Open the database file *file* and return a corresponding object.

If the database file already exists, the *whichdb()* function is used to determine its type and the appropriate module is used; if it does not exist, the first module listed above that can be imported is used.

The optional *flag* argument can be:

Valor	Significado
`'r'`	Open existing database for reading only (default)
`'w'`	Open existing database for reading and writing
`'c'`	Open database for reading and writing, creating it if it doesn't exist
`'n'`	Always create a new, empty database, open for reading and writing

The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal `0o666` (and will be modified by the prevailing umask).

The object returned by *open()* supports the same basic functionality as dictionaries; keys and their corresponding values can be stored, retrieved, and deleted, and the `in` operator and the `keys()` method are available, as well as `get()` and `setdefault()`.

Alterado na versão 3.2: `get()` and `setdefault()` are now available in all database modules.

Key and values are always stored as bytes. This means that when strings are used they are implicitly converted to the default encoding before being stored.

These objects also support being used in a `with` statement, which will automatically close them when done.

Alterado na versão 3.4: Added native support for the context management protocol to the objects returned by *open()*.

The following example records some hostnames and a corresponding title, and then prints out the contents of the database:

```
import dbm

# Open database, creating it if necessary.
with dbm.open('cache', 'c') as db:

    # Record some values
    db[b'hello'] = b'there'
    db['www.python.org'] = 'Python Website'
    db['www.cnn.com'] = 'Cable News Network'

    # Note that the keys are considered bytes now.
    assert db[b'www.python.org'] == b'Python Website'
    # Notice how the value is now in bytes.
    assert db['www.cnn.com'] == b'Cable News Network'

    # Often-used methods of the dict interface work too.
    print(db.get('python.org', b'not present'))

    # Storing a non-string key or value will raise an exception (most
```

(continua na próxima página)

(continuação da página anterior)

```
    # likely a TypeError).
    db['www.yahoo.com'] = 4

# db is automatically closed when leaving the with statement.
```

Ver também:

Módulo ***shelve*** Persistence module which stores non-string data.

The individual submodules are described in the following sections.

12.5.1 dbm.gnu — GNU's reinterpretation of dbm

Source code: Lib/dbm/gnu.py

This module is quite similar to the *dbm* module, but uses the GNU library gdbm instead to provide some additional functionality. Please note that the file formats created by *dbm.gnu* and *dbm.ndbm* are incompatible.

The *dbm.gnu* module provides an interface to the GNU DBM library. dbm.gnu.gdbm objects behave like mappings (dictionaries), except that keys and values are always converted to bytes before storing. Printing a gdbm object doesn't print the keys and values, and the items() and values() methods are not supported.

exception dbm.gnu.**error**
Raised on *dbm.gnu*-specific errors, such as I/O errors. *KeyError* is raised for general mapping errors like specifying an incorrect key.

dbm.gnu.**open**(*filename*[, *flag*[, *mode*]])
Open a gdbm database and return a gdbm object. The *filename* argument is the name of the database file.

The optional *flag* argument can be:

Valor	Significado
'r'	Open existing database for reading only (default)
'w'	Open existing database for reading and writing
'c'	Open database for reading and writing, creating it if it doesn't exist
'n'	Always create a new, empty database, open for reading and writing

The following additional characters may be appended to the flag to control how the database is opened:

Valor	Significado
'f'	Open the database in fast mode. Writes to the database will not be synchronized.
's'	Synchronized mode. This will cause changes to the database to be immediately written to the file.
'u'	Do not lock database.

Not all flags are valid for all versions of gdbm. The module constant open_flags is a string of supported flag characters. The exception *error* is raised if an invalid flag is specified.

The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal 0o666.

In addition to the dictionary-like methods, gdbm objects have the following methods:

gdbm.**firstkey**()
It's possible to loop over every key in the database using this method and the *nextkey()* method. The traversal is ordered by gdbm's internal hash values, and won't be sorted by the key values. This method returns the starting key.

gdbm.**nextkey**(*key*)
Returns the key that follows *key* in the traversal. The following code prints every key in the database db, without having to create a list in memory that contains them all:

```
k = db.firstkey()
while k != None:
    print(k)
    k = db.nextkey(k)
```

gdbm.**reorganize**()
If you have carried out a lot of deletions and would like to shrink the space used by the gdbm file, this routine will reorganize the database. gdbm objects will not shorten the length of a database file except by using this reorganization; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

gdbm.**sync**()
When the database has been opened in fast mode, this method forces any unwritten data to be written to the disk.

gdbm.**close**()
Fecha o banco de dados gdbm.

12.5.2 dbm.ndbm — Interface based on ndbm

Source code: Lib/dbm/ndbm.py

The *dbm.ndbm* module provides an interface to the Unix "(n)dbm" library. Dbm objects behave like mappings (dictionaries), except that keys and values are always stored as bytes. Printing a dbm object doesn't print the keys and values, and the items() and values() methods are not supported.

This module can be used with the "classic" ndbm interface or the GNU GDBM compatibility interface. On Unix, the **configure** script will attempt to locate the appropriate header file to simplify building this module.

exception dbm.ndbm.**error**
Raised on *dbm.ndbm*-specific errors, such as I/O errors. *KeyError* is raised for general mapping errors like specifying an incorrect key.

dbm.ndbm.**library**
Name of the ndbm implementation library used.

dbm.ndbm.**open**(*filename*[, *flag*[, *mode*]])
Open a dbm database and return a ndbm object. The *filename* argument is the name of the database file (without the .dir or .pag extensions).

The optional *flag* argument must be one of these values:

Valor	Significado
'r'	Open existing database for reading only (default)
'w'	Open existing database for reading and writing
'c'	Open database for reading and writing, creating it if it doesn't exist
'n'	Always create a new, empty database, open for reading and writing

The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal `0o666` (and will be modified by the prevailing umask).

In addition to the dictionary-like methods, `ndbm` objects provide the following method:

`ndbm.close()`
Fecha o banco de dados `ndbm`.

12.5.3 `dbm.dumb` — Portable DBM implementation

Source code: Lib/dbm/dumb.py

Nota: The *dbm.dumb* module is intended as a last resort fallback for the *dbm* module when a more robust module is not available. The *dbm.dumb* module is not written for speed and is not nearly as heavily used as the other database modules.

The *dbm.dumb* module provides a persistent dictionary-like interface which is written entirely in Python. Unlike other modules such as *dbm.gnu* no external library is required. As with other persistent mappings, the keys and values are always stored as bytes.

O módulo define o seguinte:

exception `dbm.dumb.error`
Raised on *dbm.dumb*-specific errors, such as I/O errors. *KeyError* is raised for general mapping errors like specifying an incorrect key.

`dbm.dumb.open`(*filename*[, *flag*[, *mode*]])
Open a `dumbdbm` database and return a dumbdbm object. The *filename* argument is the basename of the database file (without any specific extensions). When a dumbdbm database is created, files with `.dat` and `.dir` extensions are created.

The optional *flag* argument supports only the semantics of `'c'` and `'n'` values. Other values will default to database being always opened for update, and will be created if it does not exist.

The optional *mode* argument is the Unix mode of the file, used only when the database has to be created. It defaults to octal `0o666` (and will be modified by the prevailing umask).

> **Aviso:** It is possible to crash the Python interpreter when loading a database with a sufficiently large/complex entry due to stack depth limitations in Python's AST compiler.

Alterado na versão 3.5: *open()* always creates a new database when the flag has the value `'n'`.

Deprecated since version 3.6, will be removed in version 3.8: Creating database in `'r'` and `'w'` modes. Modifying database in `'r'` mode.

In addition to the methods provided by the *collections.abc.MutableMapping* class, `dumbdbm` objects provide the following methods:

`dumbdbm.sync()`
Synchronize the on-disk directory and data files. This method is called by the `Shelve.sync()` method.

`dumbdbm.close()`
Close the `dumbdbm` database.

12.6 `sqlite3` — Interface DB-API 2.0 para bancos de dados SQLite

Código fonte

SQLite é uma biblioteca C que fornece um banco de dados leve baseado em disco que não requer um processo de servidor separado e permite acessar o banco de dados usando uma variante não padrão da linguagem de consulta SQL. Algumas aplicações podem usar SQLite para armazenamento interno de dados. Também é possível prototipar um aplicativo usando SQLite e, em seguida, portar o código para um banco de dados maior, como PostgreSQL ou Oracle.

The sqlite3 module was written by Gerhard Häring. It provides a SQL interface compliant with the DB-API 2.0 specification described by **PEP 249**.

To use the module, you must first create a *Connection* object that represents the database. Here the data will be stored in the `example.db` file:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

You can also supply the special name `:memory:` to create a database in RAM.

Once you have a *Connection*, you can create a *Cursor* object and call its *execute()* method to perform SQL commands:

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
             (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()
```

The data you've saved is persistent and is available in subsequent sessions:

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
```

Usually your SQL operations will need to use values from Python variables. You shouldn't assemble your query using Python's string operations because doing so is insecure; it makes your program vulnerable to an SQL injection attack (see https://xkcd.com/327/ for humorous example of what can go wrong).

Instead, use the DB-API's parameter substitution. Put `?` as a placeholder wherever you want to use a value, and then provide a tuple of values as the second argument to the cursor's *execute()* method. (Other database modules may use a different placeholder, such as `%s` or `:1`.) For example:

```
# Never do this -- insecure!
symbol = 'RHAT'
c.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)
```

(continua na próxima página)

(continuação da página anterior)

```
# Do this instead
t = ('RHAT',)
c.execute('SELECT * FROM stocks WHERE symbol=?', t)
print(c.fetchone())

# Larger example that inserts many records at a time
purchases = [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
             ('2006-04-05', 'BUY', 'MSFT', 1000, 72.00),
             ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
            ]
c.executemany('INSERT INTO stocks VALUES (?,?,?,?,?)', purchases)
```

To retrieve data after executing a SELECT statement, you can either treat the cursor as an *iterator*, call the cursor's `fetchone()` method to retrieve a single matching row, or call `fetchall()` to get a list of the matching rows.

This example uses the iterator form:

```
>>> for row in c.execute('SELECT * FROM stocks ORDER BY price'):
        print(row)

('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
('2006-04-06', 'SELL', 'IBM', 500, 53.0)
('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

Ver também:

https://github.com/ghaering/pysqlite The pysqlite web page – sqlite3 is developed externally under the name "pysqlite".

https://www.sqlite.org A página web do SQLite; a documentação descreve a sintaxe e os tipos de dados disponíveis para o dialeto SQL suportado.

https://www.w3schools.com/sql/ Tutoriais, referências e exemplos para aprender a sintaxe SQL.

PEP 249 - Especificação 2.0 da API de banco de dados PEP escrita por Marc-André Lemburg.

12.6.1 Funções e constantes do módulo

`sqlite3.`**`version`**
The version number of this module, as a string. This is not the version of the SQLite library.

`sqlite3.`**`version_info`**
The version number of this module, as a tuple of integers. This is not the version of the SQLite library.

`sqlite3.`**`sqlite_version`**
The version number of the run-time SQLite library, as a string.

`sqlite3.`**`sqlite_version_info`**
The version number of the run-time SQLite library, as a tuple of integers.

`sqlite3.`**`PARSE_DECLTYPES`**
This constant is meant to be used with the *detect_types* parameter of the `connect()` function.

Setting it makes the `sqlite3` module parse the declared type for each column it returns. It will parse out the first word of the declared type, i. e. for "integer primary key", it will parse out "integer", or for "number(10)" it will parse out "number". Then for that column, it will look into the converters dictionary and use the converter function registered for that type there.

sqlite3.**PARSE_COLNAMES**
This constant is meant to be used with the *detect_types* parameter of the `connect()` function.

Setting this makes the SQLite interface parse the column name for each column it returns. It will look for a string formed [mytype] in there, and then decide that 'mytype' is the type of the column. It will try to find an entry of 'mytype' in the converters dictionary and then use the converter function found there to return the value. The column name found in `Cursor.description` does not include the type, i. e. if you use something like `'as "Expiration date [datetime]"'` in your SQL, then we will parse out everything until the first `'['` for the column name and strip the preceeding space: the column name would simply be "Expiration date".

sqlite3.**connect**(*database*[, *timeout*, *detect_types*, *isolation_level*, *check_same_thread*, *factory*, *cached_statements*, *uri*])
Opens a connection to the SQLite database file *database*. By default returns a `Connection` object, unless a custom *factory* is given.

database is a *path-like object* giving the pathname (absolute or relative to the current working directory) of the database file to be opened. You can use `":memory:"` to open a database connection to a database that resides in RAM instead of on disk.

When a database is accessed by multiple connections, and one of the processes modifies the database, the SQLite database is locked until that transaction is committed. The *timeout* parameter specifies how long the connection should wait for the lock to go away until raising an exception. The default for the timeout parameter is 5.0 (five seconds).

For the *isolation_level* parameter, please see the `isolation_level` property of `Connection` objects.

SQLite natively supports only the types TEXT, INTEGER, REAL, BLOB and NULL. If you want to use other types you must add support for them yourself. The *detect_types* parameter and the using custom **converters** registered with the module-level `register_converter()` function allow you to easily do that.

detect_types defaults to 0 (i. e. off, no type detection), you can set it to any combination of `PARSE_DECLTYPES` and `PARSE_COLNAMES` to turn type detection on.

By default, *check_same_thread* is `True` and only the creating thread may use the connection. If set `False`, the returned connection may be shared across multiple threads. When using multiple threads with the same connection writing operations should be serialized by the user to avoid data corruption.

By default, the `sqlite3` module uses its `Connection` class for the connect call. You can, however, subclass the `Connection` class and make `connect()` use your class instead by providing your class for the *factory* parameter.

Consult the section *SQLite and Python types* of this manual for details.

The `sqlite3` module internally uses a statement cache to avoid SQL parsing overhead. If you want to explicitly set the number of statements that are cached for the connection, you can set the *cached_statements* parameter. The currently implemented default is to cache 100 statements.

If *uri* is true, *database* is interpreted as a URI. This allows you to specify options. For example, to open a database in read-only mode you can use:

```
db = sqlite3.connect('file:path/to/database?mode=ro', uri=True)
```

More information about this feature, including a list of recognized options, can be found in the SQLite URI documentation.

Alterado na versão 3.4: Adicionado o parâmetro *uri*.

Alterado na versão 3.7: *database* can now also be a *path-like object*, not only a string.

sqlite3.**register_converter**(*typename*, *callable*)
Registers a callable to convert a bytestring from the database into a custom Python type. The callable will be invoked for all database values that are of the type *typename*. Confer the parameter *detect_types* of the `connect()`

function for how the type detection works. Note that *typename* and the name of the type in your query are matched in case-insensitive manner.

sqlite3.**register_adapter**(*type*, *callable*)
Registers a callable to convert the custom Python type *type* into one of SQLite's supported types. The callable *callable* accepts as single parameter the Python value, and must return a value of the following types: int, float, str or bytes.

sqlite3.**complete_statement**(*sql*)
Returns `True` if the string *sql* contains one or more complete SQL statements terminated by semicolons. It does not verify that the SQL is syntactically correct, only that there are no unclosed string literals and the statement is terminated by a semicolon.

This can be used to build a shell for SQLite, as in the following example:

```
# A minimal SQLite shell for experiments

import sqlite3

con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()

buffer = ""

print("Enter your SQL commands to execute in sqlite3.")
print("Enter a blank line to exit.")

while True:
    line = input()
    if line == "":
        break
    buffer += line
    if sqlite3.complete_statement(buffer):
        try:
            buffer = buffer.strip()
            cur.execute(buffer)

            if buffer.lstrip().upper().startswith("SELECT"):
                print(cur.fetchall())
        except sqlite3.Error as e:
            print("An error occurred:", e.args[0])
        buffer = ""

con.close()
```

sqlite3.**enable_callback_tracebacks**(*flag*)
By default you will not get any tracebacks in user-defined functions, aggregates, converters, authorizer callbacks etc. If you want to debug them, you can call this function with *flag* set to `True`. Afterwards, you will get tracebacks from callbacks on `sys.stderr`. Use `False` to disable the feature again.

12.6.2 Connection Objects

class sqlite3.**Connection**

A SQLite database connection has the following attributes and methods:

isolation_level

Get or set the current default isolation level. `None` for autocommit mode or one of "DEFERRED", "IMMEDIATE" or "EXCLUSIVE". See section *Controlando Transações* for a more detailed explanation.

in_transaction

`True` if a transaction is active (there are uncommitted changes), `False` otherwise. Read-only attribute.

Novo na versão 3.2.

cursor(*factory=Cursor*)

The cursor method accepts a single optional parameter *factory*. If supplied, this must be a callable returning an instance of `Cursor` or its subclasses.

commit()

This method commits the current transaction. If you don't call this method, anything you did since the last call to `commit()` is not visible from other database connections. If you wonder why you don't see the data you've written to the database, please check you didn't forget to call this method.

rollback()

This method rolls back any changes to the database since the last call to `commit()`.

close()

This closes the database connection. Note that this does not automatically call `commit()`. If you just close your database connection without calling `commit()` first, your changes will be lost!

execute(*sql*[, *parameters*])

This is a nonstandard shortcut that creates a cursor object by calling the `cursor()` method, calls the cursor's `execute()` method with the *parameters* given, and returns the cursor.

executemany(*sql*[, *parameters*])

This is a nonstandard shortcut that creates a cursor object by calling the `cursor()` method, calls the cursor's `executemany()` method with the *parameters* given, and returns the cursor.

executescript(*sql_script*)

This is a nonstandard shortcut that creates a cursor object by calling the `cursor()` method, calls the cursor's `executescript()` method with the given *sql_script*, and returns the cursor.

create_function(*name*, *num_params*, *func*)

Creates a user-defined function that you can later use from within SQL statements under the function name *name*. *num_params* is the number of parameters the function accepts (if *num_params* is -1, the function may take any number of arguments), and *func* is a Python callable that is called as the SQL function.

The function can return any of the types supported by SQLite: bytes, str, int, float and `None`.

Exemplo:

```
import sqlite3
import hashlib

def md5sum(t):
    return hashlib.md5(t).hexdigest()

con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", (b"foo",))
```

(continua na próxima página)

(continuação da página anterior)

```
print(cur.fetchone()[0])

con.close()
```

create_aggregate (*name*, *num_params*, *aggregate_class*)
Creates a user-defined aggregate function.

The aggregate class must implement a `step` method, which accepts the number of parameters *num_params* (if *num_params* is -1, the function may take any number of arguments), and a `finalize` method which will return the final result of the aggregate.

The `finalize` method can return any of the types supported by SQLite: bytes, str, int, float and `None`.

Exemplo:

```
import sqlite3

class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.cursor()
cur.execute("create table test(i)")
cur.execute("insert into test(i) values (1)")
cur.execute("insert into test(i) values (2)")
cur.execute("select mysum(i) from test")
print(cur.fetchone()[0])

con.close()
```

create_collation (*name*, *callable*)
Creates a collation with the specified *name* and *callable*. The callable will be passed two string arguments. It should return -1 if the first is ordered lower than the second, 0 if they are ordered equal and 1 if the first is ordered higher than the second. Note that this controls sorting (ORDER BY in SQL) so your comparisons don't affect other SQL operations.

Note that the callable will get its parameters as Python bytestrings, which will normally be encoded in UTF-8.

The following example shows a custom collation that sorts "the wrong way":

```
import sqlite3

def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
        return 1
    else:
        return -1
```

(continua na próxima página)

(continuação da página anterior)

```
con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.cursor()
cur.execute("create table test(x)")
cur.executemany("insert into test(x) values (?)", [("a",), ("b",)])
cur.execute("select x from test order by x collate reverse")
for row in cur:
    print(row)
con.close()
```

To remove a collation, call `create_collation` with `None` as callable:

```
con.create_collation("reverse", None)
```

interrupt()
You can call this method from a different thread to abort any queries that might be executing on the connection. The query will then abort and the caller will get an exception.

set_authorizer(*authorizer_callback*)
This routine registers a callback. The callback is invoked for each attempt to access a column of a table in the database. The callback should return `SQLITE_OK` if access is allowed, `SQLITE_DENY` if the entire SQL statement should be aborted with an error and `SQLITE_IGNORE` if the column should be treated as a NULL value. These constants are available in the *sqlite3* module.

The first argument to the callback signifies what kind of operation is to be authorized. The second and third argument will be arguments or *None* depending on the first argument. The 4th argument is the name of the database ("main", "temp", etc.) if applicable. The 5th argument is the name of the inner-most trigger or view that is responsible for the access attempt or *None* if this access attempt is directly from input SQL code.

Please consult the SQLite documentation about the possible values for the first argument and the meaning of the second and third argument depending on the first one. All necessary constants are available in the *sqlite3* module.

set_progress_handler(*handler*, *n*)
This routine registers a callback. The callback is invoked for every *n* instructions of the SQLite virtual machine. This is useful if you want to get called from SQLite during long-running operations, for example to update a GUI.

If you want to clear any previously installed progress handler, call the method with *None* for *handler*.

Returning a non-zero value from the handler function will terminate the currently executing query and cause it to raise an *OperationalError* exception.

set_trace_callback(*trace_callback*)
Registers *trace_callback* to be called for each SQL statement that is actually executed by the SQLite backend.

The only argument passed to the callback is the statement (as string) that is being executed. The return value of the callback is ignored. Note that the backend does not only run statements passed to the *Cursor.execute()* methods. Other sources include the transaction management of the Python module and the execution of triggers defined in the current database.

Passing *None* as *trace_callback* will disable the trace callback.

Novo na versão 3.3.

enable_load_extension(*enabled*)
This routine allows/disallows the SQLite engine to load SQLite extensions from shared libraries. SQLite

extensions can define new functions, aggregates or whole new virtual table implementations. One well-known extension is the fulltext-search extension distributed with SQLite.

Loadable extensions are disabled by default. See[1].

Novo na versão 3.2.

```
import sqlite3

con = sqlite3.connect(":memory:")

# enable extension loading
con.enable_load_extension(True)

# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using an API call:
# con.load_extension("./fts3.so")

# disable extension loading again
con.enable_load_extension(False)

# example from SQLite wiki
con.execute("create virtual table recipe using fts3(name, ingredients)")
con.executescript("""
    insert into recipe (name, ingredients) values ('broccoli stew', 'broccoli␣
↪peppers cheese tomatoes');
    insert into recipe (name, ingredients) values ('pumpkin stew', 'pumpkin␣
↪onions garlic celery');
    insert into recipe (name, ingredients) values ('broccoli pie', 'broccoli␣
↪cheese onions flour');
    insert into recipe (name, ingredients) values ('pumpkin pie', 'pumpkin␣
↪sugar flour butter');
    """)
for row in con.execute("select rowid, name, ingredients from recipe where␣
↪name match 'pie'"):
    print(row)

con.close()
```

load_extension(*path*)
This routine loads a SQLite extension from a shared library. You have to enable extension loading with *enable_load_extension()* before you can use this routine.

Loadable extensions are disabled by default. See[1].

Novo na versão 3.2.

row_factory
You can change this attribute to a callable that accepts the cursor and the original row as a tuple and will return the real result row. This way, you can implement more advanced ways of returning results, such as returning an object that can also access columns by name.

Exemplo:

[1] The sqlite3 module is not built with loadable extension support by default, because some platforms (notably Mac OS X) have SQLite libraries which are compiled without this feature. To get loadable extension support, you must pass –enable-loadable-sqlite-extensions to configure.

```
import sqlite3

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d

con = sqlite3.connect(":memory:")
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print(cur.fetchone()["a"])

con.close()
```

If returning a tuple doesn't suffice and you want name-based access to columns, you should consider setting *row_factory* to the highly-optimized *sqlite3.Row* type. *Row* provides both index-based and case-insensitive name-based access to columns with almost no memory overhead. It will probably be better than your own custom dictionary-based approach or even a db_row based solution.

text_factory

Using this attribute you can control what objects are returned for the TEXT data type. By default, this attribute is set to *str* and the *sqlite3* module will return Unicode objects for TEXT. If you want to return bytestrings instead, you can set it to *bytes*.

You can also set it to any other callable that accepts a single bytestring parameter and returns the resulting object.

See the following example code for illustration:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()

AUSTRIA = "\xd6sterreich"

# by default, rows are returned as Unicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert row[0] == AUSTRIA

# but we can make sqlite3 always return bytestrings ...
con.text_factory = bytes
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is bytes
# the bytestrings will be encoded in UTF-8, unless you stored garbage in the
# database ...
assert row[0] == AUSTRIA.encode("utf-8")

# we can also implement a custom text_factory ...
# here we implement one that appends "foo" to all strings
con.text_factory = lambda x: x.decode("utf-8") + "foo"
cur.execute("select ?", ("bar",))
row = cur.fetchone()
assert row[0] == "barfoo"
```

(continua na próxima página)

(continuação da página anterior)

```
con.close()
```

total_changes
Returns the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.

iterdump()
Returns an iterator to dump the database in an SQL text format. Useful when saving an in-memory database for later restoration. This function provides the same capabilities as the `.dump` command in the **sqlite3** shell.

Exemplo:

```
# Convert file existing_db.db to SQL dump file dump.sql
import sqlite3

con = sqlite3.connect('existing_db.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
con.close()
```

backup(*target*, ***, *pages=0*, *progress=None*, *name="main"*, *sleep=0.250*)
This method makes a backup of a SQLite database even while it's being accessed by other clients, or concurrently by the same connection. The copy will be written into the mandatory argument *target*, that must be another *Connection* instance.

By default, or when *pages* is either `0` or a negative integer, the entire database is copied in a single step; otherwise the method performs a loop copying up to *pages* pages at a time.

If *progress* is specified, it must either be `None` or a callable object that will be executed at each iteration with three integer arguments, respectively the *status* of the last iteration, the *remaining* number of pages still to be copied and the *total* number of pages.

The *name* argument specifies the database name that will be copied: it must be a string containing either `"main"`, the default, to indicate the main database, `"temp"` to indicate the temporary database or the name specified after the `AS` keyword in an `ATTACH DATABASE` statement for an attached database.

The *sleep* argument specifies the number of seconds to sleep by between successive attempts to backup remaining pages, can be specified either as an integer or a floating point value.

Example 1, copy an existing database into another:

```
import sqlite3

def progress(status, remaining, total):
    print(f'Copied {total-remaining} of {total} pages...')

con = sqlite3.connect('existing_db.db')
bck = sqlite3.connect('backup.db')
with bck:
    con.backup(bck, pages=1, progress=progress)
bck.close()
con.close()
```

Example 2, copy an existing database into a transient copy:

```
import sqlite3

source = sqlite3.connect('existing_db.db')
dest = sqlite3.connect(':memory:')
source.backup(dest)
```

Availability: SQLite 3.6.11 or higher

Novo na versão 3.7.

12.6.3 Cursor Objects

class sqlite3.**Cursor**

A *Cursor* instance has the following attributes and methods.

execute(*sql*[, *parameters*])

Executes an SQL statement. The SQL statement may be parameterized (i. e. placeholders instead of SQL literals). The *sqlite3* module supports two kinds of placeholders: question marks (qmark style) and named placeholders (named style).

Here's an example of both styles:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table people (name_last, age)")

who = "Yeltsin"
age = 72

# This is the qmark style:
cur.execute("insert into people values (?, ?)", (who, age))

# And this is the named style:
cur.execute("select * from people where name_last=:who and age=:age", {"who":␣
↪who, "age": age})

print(cur.fetchone())

con.close()
```

execute() will only execute a single SQL statement. If you try to execute more than one statement with it, it will raise a *Warning*. Use *executescript()* if you want to execute multiple SQL statements with one call.

executemany(*sql*, *seq_of_parameters*)

Executes an SQL command against all parameter sequences or mappings found in the sequence *seq_of_parameters*. The *sqlite3* module also allows using an *iterator* yielding parameters instead of a sequence.

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')
```

(continua na próxima página)

(continuação da página anterior)

```
    def __iter__(self):
        return self

    def __next__(self):
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),) # this is a 1-tuple

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

cur.execute("select c from characters")
print(cur.fetchall())

con.close()
```

Here's a shorter example using a *generator*:

```
import sqlite3
import string

def char_generator():
    for c in string.ascii_lowercase:
        yield (c,)

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

cur.executemany("insert into characters(c) values (?)", char_generator())

cur.execute("select c from characters")
print(cur.fetchall())

con.close()
```

executescript(*sql_script*)

This is a nonstandard convenience method for executing multiple SQL statements at once. It issues a `COMMIT` statement first, then executes the SQL script it gets as a parameter.

sql_script can be an instance of *str*.

Exemplo:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
    create table person(
        firstname,
```

(continua na próxima página)

(continuação da página anterior)

```
        lastname,
        age
    );

    create table book(
        title,
        author,
        published
    );

    insert into book(title, author, published)
    values (
        'Dirk Gently''s Holistic Detective Agency',
        'Douglas Adams',
        1987
    );
    """)
con.close()
```

fetchone()
Fetches the next row of a query result set, returning a single sequence, or *None* when no more data is available.

fetchmany(*size=cursor.arraysize*)
Fetches the next set of rows of a query result, returning a list. An empty list is returned when no more rows are available.

The number of rows to fetch per call is specified by the *size* parameter. If it is not given, the cursor's arraysize determines the number of rows to be fetched. The method should try to fetch as many rows as indicated by the size parameter. If this is not possible due to the specified number of rows not being available, fewer rows may be returned.

Note there are performance considerations involved with the *size* parameter. For optimal performance, it is usually best to use the arraysize attribute. If the *size* parameter is used, then it is best for it to retain the same value from one *fetchmany()* call to the next.

fetchall()
Fetches all (remaining) rows of a query result, returning a list. Note that the cursor's arraysize attribute can affect the performance of this operation. An empty list is returned when no rows are available.

close()
Close the cursor now (rather than whenever __del__ is called).

The cursor will be unusable from this point forward; a *ProgrammingError* exception will be raised if any operation is attempted with the cursor.

rowcount
Although the *Cursor* class of the *sqlite3* module implements this attribute, the database engine's own support for the determination of "rows affected"/"rows selected" is quirky.

For *executemany()* statements, the number of modifications are summed up into *rowcount*.

As required by the Python DB API Spec, the *rowcount* attribute "is -1 in case no executeXX() has been performed on the cursor or the rowcount of the last operation is not determinable by the interface". This includes SELECT statements because we cannot determine the number of rows a query produced until all rows were fetched.

With SQLite versions before 3.6.5, *rowcount* is set to 0 if you make a DELETE FROM table without any condition.

lastrowid
This read-only attribute provides the rowid of the last modified row. It is only set if you issued an `INSERT` or a `REPLACE` statement using the *execute()* method. For operations other than `INSERT` or `REPLACE` or when *executemany()* is called, *lastrowid* is set to *None*.

If the `INSERT` or `REPLACE` statement failed to insert the previous successful rowid is returned.

Alterado na versão 3.6: Added support for the `REPLACE` statement.

arraysize
Read/write attribute that controls the number of rows returned by *fetchmany()*. The default value is 1 which means a single row would be fetched per call.

description
This read-only attribute provides the column names of the last query. To remain compatible with the Python DB API, it returns a 7-tuple for each column where the last six items of each tuple are *None*.

It is set for `SELECT` statements without any matching rows as well.

connection
This read-only attribute provides the SQLite database *Connection* used by the *Cursor* object. A *Cursor* object created by calling *con.cursor()* will have a *connection* attribute that refers to *con*:

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
```

12.6.4 Row Objects

class sqlite3.**Row**
A *Row* instance serves as a highly optimized *row_factory* for *Connection* objects. It tries to mimic a tuple in most of its features.

It supports mapping access by column name and index, iteration, representation, equality testing and *len()*.

If two *Row* objects have exactly the same columns and their members are equal, they compare equal.

keys()
This method returns a list of column names. Immediately after a query, it is the first member of each tuple in *Cursor.description*.

Alterado na versão 3.5: Added support of slicing.

Let's assume we initialize a table as in the example given above:

```
conn = sqlite3.connect(":memory:")
c = conn.cursor()
c.execute('''create table stocks
(date text, trans text, symbol text,
 qty real, price real)''')
c.execute("""insert into stocks
          values ('2006-01-05','BUY','RHAT',100,35.14)""")
conn.commit()
c.close()
```

Now we plug *Row* in:

```
>>> conn.row_factory = sqlite3.Row
>>> c = conn.cursor()
>>> c.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = c.fetchone()
>>> type(r)
<class 'sqlite3.Row'>
>>> tuple(r)
('2006-01-05', 'BUY', 'RHAT', 100.0, 35.14)
>>> len(r)
5
>>> r[2]
'RHAT'
>>> r.keys()
['date', 'trans', 'symbol', 'qty', 'price']
>>> r['qty']
100.0
>>> for member in r:
...     print(member)
...
2006-01-05
BUY
RHAT
100.0
35.14
```

12.6.5 Exceções

exception sqlite3.**Warning**
A subclass of *Exception*.

exception sqlite3.**Error**
A classe base das outras exceções neste módulo. É uma subclasse de *Exception*.

exception sqlite3.**DatabaseError**
Exception raised for errors that are related to the database.

exception sqlite3.**IntegrityError**
Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails. It is a subclass of *DatabaseError*.

exception sqlite3.**ProgrammingError**
Exception raised for programming errors, e.g. table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, etc. It is a subclass of *DatabaseError*.

exception sqlite3.**OperationalError**
Exception raised for errors that are related to the database's operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, the data source name is not found, a transaction could not be processed, etc. It is a subclass of *DatabaseError*.

exception sqlite3.**NotSupportedError**
Exception raised in case a method or database API was used which is not supported by the database, e.g. calling the *rollback()* method on a connection that does not support transaction or has transactions turned off. It is a subclass of *DatabaseError*.

12.6.6 SQLite and Python types

Introdução

SQLite natively supports the following types: `NULL`, `INTEGER`, `REAL`, `TEXT`, `BLOB`.

The following Python types can thus be sent to SQLite without any problem:

Python type	Tipo SQLite
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

This is how SQLite types are converted to Python types by default:

Tipo SQLite	Python type
NULL	*None*
INTEGER	*int*
REAL	*float*
TEXT	depends on *text_factory*, *str* by default
BLOB	*bytes*

The type system of the *sqlite3* module is extensible in two ways: you can store additional Python types in a SQLite database via object adaptation, and you can let the *sqlite3* module convert SQLite types to different Python types via converters.

Using adapters to store additional Python types in SQLite databases

As described before, SQLite supports only a limited set of types natively. To use other Python types with SQLite, you must **adapt** them to one of the sqlite3 module's supported types for SQLite: one of NoneType, int, float, str, bytes.

There are two ways to enable the *sqlite3* module to adapt a custom Python type to one of the supported ones.

Letting your object adapt itself

This is a good approach if you write the class yourself. Let's suppose you have a class like this:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

Now you want to store the point in a single SQLite column. First you'll have to choose one of the supported types first to be used for representing the point. Let's just use str and separate the coordinates using a semicolon. Then you need to give your class a method `__conform__(self, protocol)` which must return the converted value. The parameter *protocol* will be `PrepareProtocol`.

```
import sqlite3

class Point:
    def __init__(self, x, y):
```

(continua na próxima página)

(continuação da página anterior)

```
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "%f;%f" % (self.x, self.y)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])

con.close()
```

Registering an adapter callable

The other possibility is to create a function that converts the type to the string representation and register the function with *register_adapter()*.

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])

con.close()
```

The *sqlite3* module has two default adapters for Python's built-in *datetime.date* and *datetime.datetime* types. Now let's suppose we want to store *datetime.datetime* objects not in ISO representation, but as a Unix timestamp.

```
import sqlite3
import datetime
import time

def adapt_datetime(ts):
    return time.mktime(ts.timetuple())

sqlite3.register_adapter(datetime.datetime, adapt_datetime)

con = sqlite3.connect(":memory:")
```

(continua na próxima página)

(continuação da página anterior)

```
cur = con.cursor()

now = datetime.datetime.now()
cur.execute("select ?", (now,))
print(cur.fetchone()[0])

con.close()
```

Converting SQLite values to custom Python types

Writing an adapter lets you send custom Python types to SQLite. But to make it really useful we need to make the Python to SQLite to Python roundtrip work.

Enter converters.

Let's go back to the `Point` class. We stored the x and y coordinates separated via semicolons as strings in SQLite.

First, we'll define a converter function that accepts the string as a parameter and constructs a `Point` object from it.

Nota: Converter functions **always** get called with a *bytes* object, no matter under which data type you sent the value to SQLite.

```
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

Now you need to make the *sqlite3* module know that what you select from the database is actually a point. There are two ways of doing this:

- Implicitly via the declared type
- Explicitly via the column name

Both ways are described in section *Funções e constantes do módulo*, in the entries for the constants *PARSE_DECLTYPES* and *PARSE_COLNAMES*.

The following example illustrates both approaches.

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "(%f;%f)" % (self.x, self.y)

def adapt_point(point):
    return ("%f;%f" % (point.x, point.y)).encode('ascii')

def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)

# Register the adapter
```

(continua na próxima página)

(continuação da página anterior)

```
sqlite3.register_adapter(Point, adapt_point)

# Register the converter
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

########################
# 1) Using declared types
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p point)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute("select p from test")
print("with declared types:", cur.fetchone()[0])
cur.close()
con.close()

######################
# 1) Using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(p)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute('select p as "p [point]" from test')
print("with column names:", cur.fetchone()[0])
cur.close()
con.close()
```

Default adapters and converters

There are default adapters for the date and datetime types in the datetime module. They will be sent as ISO dates/ISO timestamps to SQLite.

The default converters are registered under the name "date" for *datetime.date* and under the name "timestamp" for *datetime.datetime*.

This way, you can use date/timestamps from Python without any additional fiddling in most cases. The format of the adapters is also compatible with the experimental SQLite date/time functions.

The following example demonstrates this.

```
import sqlite3
import datetime

con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_
↪COLNAMES)
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)")

today = datetime.date.today()
now = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
```

(continua na próxima página)

(continuação da página anterior)

```
cur.execute("select d, ts from test")
row = cur.fetchone()
print(today, "=>", row[0], type(row[0]))
print(now, "=>", row[1], type(row[1]))

cur.execute('select current_date as "d [date]", current_timestamp as "ts [timestamp]"
↪')
row = cur.fetchone()
print("current_date", row[0], type(row[0]))
print("current_timestamp", row[1], type(row[1]))

con.close()
```

If a timestamp stored in SQLite has a fractional part longer than 6 numbers, its value will be truncated to microsecond precision by the timestamp converter.

12.6.7 Controlando Transações

The underlying `sqlite3` library operates in `autocommit` mode by default, but the Python *sqlite3* module by default does not.

`autocommit` mode means that statements that modify the database take effect immediately. A `BEGIN` or `SAVEPOINT` statement disables `autocommit` mode, and a `COMMIT`, a `ROLLBACK`, or a `RELEASE` that ends the outermost transaction, turns `autocommit` mode back on.

The Python *sqlite3* module by default issues a `BEGIN` statement implicitly before a Data Modification Language (DML) statement (i.e. `INSERT`/`UPDATE`/`DELETE`/`REPLACE`).

You can control which kind of `BEGIN` statements *sqlite3* implicitly executes via the *isolation_level* parameter to the *connect()* call, or via the `isolation_level` property of connections. If you specify no *isolation_level*, a plain `BEGIN` is used, which is equivalent to specifying `DEFERRED`. Other possible values are `IMMEDIATE` and `EXCLUSIVE`.

You can disable the *sqlite3* module's implicit transaction management by setting `isolation_level` to `None`. This will leave the underlying `sqlite3` library operating in `autocommit` mode. You can then completely control the transaction state by explicitly issuing `BEGIN`, `ROLLBACK`, `SAVEPOINT`, and `RELEASE` statements in your code.

Alterado na versão 3.6: *sqlite3* used to implicitly commit an open transaction before DDL statements. This is no longer the case.

12.6.8 Using `sqlite3` efficiently

Using shortcut methods

Using the nonstandard `execute()`, `executemany()` and `executescript()` methods of the *Connection* object, your code can be written more concisely because you don't have to create the (often superfluous) *Cursor* objects explicitly. Instead, the *Cursor* objects are created implicitly and these shortcut methods return the cursor objects. This way, you can execute a `SELECT` statement and iterate over it directly using only a single call on the *Connection* object.

```
import sqlite3

persons = [
    ("Hugo", "Boss"),
    ("Calvin", "Klein")
    ]
```

(continua na próxima página)

(continuação da página anterior)

```
con = sqlite3.connect(":memory:")

# Create the table
con.execute("create table person(firstname, lastname)")

# Fill the table
con.executemany("insert into person(firstname, lastname) values (?, ?)", persons)

# Print the table contents
for row in con.execute("select firstname, lastname from person"):
    print(row)

print("I just deleted", con.execute("delete from person").rowcount, "rows")

# close is not a shortcut method and it's not called automatically,
# so the connection object should be closed manually
con.close()
```

Accessing columns by name instead of by index

One useful feature of the *sqlite3* module is the built-in *sqlite3.Row* class designed to be used as a row factory.

Rows wrapped with this class can be accessed both by index (like tuples) and case-insensitively by name:

```
import sqlite3

con = sqlite3.connect(":memory:")
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("select 'John' as name, 42 as age")
for row in cur:
    assert row[0] == row["name"]
    assert row["name"] == row["nAmE"]
    assert row[1] == row["age"]
    assert row[1] == row["AgE"]

con.close()
```

Using the connection as a context manager

Connection objects can be used as context managers that automatically commit or rollback transactions. In the event of an exception, the transaction is rolled back; otherwise, the transaction is committed:

```
import sqlite3

con = sqlite3.connect(":memory:")
con.execute("create table person (id integer primary key, firstname varchar unique)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("insert into person(firstname) values (?)", ("Joe",))
```

(continua na próxima página)

(continuação da página anterior)

```
# con.rollback() is called after the with block finishes with an exception, the
# exception is still raised and must be caught
try:
    with con:
        con.execute("insert into person(firstname) values (?)", ("Joe",))
except sqlite3.IntegrityError:
    print("couldn't add Joe twice")

# Connection object used as context manager only commits or rollbacks transactions,
# so the connection object should be closed manually
con.close()
```

12.6.9 Common issues

Multithreading

Older SQLite versions had issues with sharing connections between threads. That's why the Python module disallows sharing connections and cursors between threads. If you still try to do so, you will get an exception at runtime.

The only exception is calling the *interrupt()* method, which only makes sense to call from a different thread.

CAPÍTULO 13

Compressão de Dados e Arquivamento

Os módulos descritos neste capítulo suportam a compressão de dados com os algoritmos zlib, gzip, bzip2 e lzma e a criação de arquivos ZIP e tar. Consulte também *Operações de arquivamento* fornecido pelo módulo `shutil`.

13.1 `zlib` — Compression compatible with `gzip`

Para aplicativos que requerem compactação de dados, as funções neste módulo permitem compactação e descompactação, usando a biblioteca zlib. A biblioteca zlib tem seu próprio site em http://www.zlib.net. Existem incompatibilidades conhecidas entre o módulo Python e as versões da biblioteca zlib anteriores à 1.1.3; O 1.1.3 possui uma vulnerabilidade de segurança, por isso recomendamos o uso do 1.1.4 ou posterior.

As funções do zlib têm muitas opções e geralmente precisam ser usadas em uma ordem específica. Esta documentação não tenta cobrir todas as permutações; consulte o manual do zlib em http://www.zlib.net/manual.html para obter informações oficiais.

Para leitura e escrita de arquivos `.gz`, consulte o módulo `gzip`.

A exceção e as funções disponíveis neste módulo são:

exception `zlib.`**error**
Exceção levantada em erros de compactação e descompactação.

`zlib.`**adler32**(*data*[, *value*])
Calcula uma soma de verificação Adler-32 de *data*. (Uma soma de verificação Adler-32 é quase tão confiável quanto uma CRC32, mas pode ser calculada muito mais rapidamente.) O resultado é um número inteiro em sinal de 32 bits. Se *value* estiver presente, ele será usado como o valor inicial da soma de verificação; caso contrário, um valor padrão de 1 é usado. A passagem de *value* permite calcular uma soma de verificação em execução através da concatenação de várias entradas. O algoritmo não é criptograficamente forte e não deve ser usado para autenticação ou assinaturas digitais. Como o algoritmo foi projetado para uso como um algoritmo de soma de verificação, não é adequado para uso como um algoritmo de hash geral.

Alterado na versão 3.0: Sempre retorna um valor sem sinal. Para gerar o mesmo valor numérico em todas as versões e plataformas do Python, use `adler32(data) & 0xffffffff`.

zlib.**compress**(*data*, *level=-1*)

Compacta os bytes em *data*, retornando um objeto de bytes contendo dados compactados. *level* é um número inteiro de `0` a `9` ou `-1` controlando o nível de compactação; `1` (Z_BEST_SPEED) é o mais rápido e produz a menor compactação, `9` (Z_BEST_COMPRESSION) é o mais lento e produz o máximo. `0` (Z_NO_COMPRESSION) é nenhuma compactação. O valor padrão é `-1` (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION representa um compromisso padrão entre velocidade e compactação (atualmente equivalente ao nível 6). Levanta a exceção *error* se ocorrer algum erro.

Alterado na versão 3.6: *level* pode agora ser usado como um parâmetro nomeado.

zlib.**compressobj**(*level=-1*, *method=DEFLATED*, *wbits=MAX_WBITS*, *memLevel=DEF_MEM_LEVEL*, *strategy=Z_DEFAULT_STRATEGY*[, *zdict*])

Retorna um objeto de compactação, a ser usado para compactar fluxos de dados que não cabem na memória de uma só vez.

level é o nível de compactação – um número inteiro de `0` a `9` ou `-1`. Um valor de `1` (Z_BEST_SPEED) é mais rápido e produz a menor compactação, enquanto um valor de `9` (Z_BEST_COMPRESSION) é mais lento e produz o máximo. `0` (Z_NO_COMPRESSION) é nenhuma compactação. O valor padrão é `-1` (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION representa um meio termo padrão entre velocidade e compactação (atualmente equivalente ao nível 6).

method é o algoritmo de compactação. Atualmente, o único valor suportado é `DEFLATED`.

O argumento *wbits* controla o tamanho do buffer do histórico (ou o "tamanho da janela") usado ao compactar dados e se um cabeçalho e um trailer estão incluídos na saída. Pode levar vários intervalos de valores, padronizando para `15` (MAX_WBITS):

- +9 a +15: o logaritmo de base dois do tamanho da janela, que varia entre 512 e 32768. Valores maiores produzem melhor compactação às custas de maior uso de memória. A saída resultante incluirá um cabeçalho e uma sequência específicos para zlib.
- -9 a -15: Usa o valor absoluto de *wbits* como o logaritmo do tamanho da janela, enquanto produz um fluxo de saída bruto sem cabeçalho ou soma de verificação à direita.
- +25 to +31 = 16 + (9 to 15): Usa os 4 bits baixos do valor como logaritmo do tamanho da janela, incluindo um cabeçalho básico **gzip** e a soma de verificação à direita na saída.

O argumento *memLevel* controla a quantidade de memória usada para o estado de compactação interno. Os valores válidos variam de `1` a `9`. Valores mais altos usam mais memória, mas são mais rápidos e produzem uma saída menor.

strategy é usado para ajustar o algoritmo de compactação. Os valores possíveis são `Z_DEFAULT_STRATEGY`, `Z_FILTERED`, `Z_HUFFMAN_ONLY`, `Z_RLE` (zlib 1.2.0.1) e `Z_FIXED` (zlib 1.2.2.2).

zdict é um dicionário de compactação predefinido. Esta é uma sequência de bytes (como um objeto *bytes*) que contém subsequências que se espera que ocorram com frequência nos dados a serem compactados. As subsequências que se espera serem mais comuns devem aparecer no final do dicionário.

Alterado na versão 3.3: Adicionado o suporte ao parâmetro e argumento nomeado *zdict*.

zlib.**crc32**(*data*[, *value*])

Calcula uma soma de verificação CRC (Cyclic Redundancy Check) de *data*. O resultado é um número inteiro sem sinal de 32 bits. Se *value* estiver presente, ele será usado como o valor inicial da soma de verificação; caso contrário, um valor padrão de 1 é usado. A passagem de *value* permite calcular uma soma de verificação em execução através da concatenação de várias entradas. O algoritmo não é criptograficamente forte e não deve ser usado para autenticação ou assinaturas digitais. Como o algoritmo foi projetado para uso como um algoritmo de soma de verificação, não é adequado para uso como um algoritmo de hash geral.

Alterado na versão 3.0: Sempre retorna um valor sem sinal. Para gerar o mesmo valor numérico em todas as versões e plataformas do Python, use `crc32(data) & 0xffffffff`.

zlib.**decompress**(*data*, *wbits=MAX_WBITS*, *bufsize=DEF_BUF_SIZE*)

Descompacta os bytes em *data*, retornando um objeto de bytes que contém os dados não compactados. O parâmetro *wbits* depende do formato de *data* e é discutido mais abaixo. Se *bufsize* for fornecido, ele será usado como o tamanho inicial do buffer de saída. Levanta a exceção `error` se ocorrer algum erro.

O parâmetro *wbits* controla o tamanho do buffer do histórico (ou "tamanho da janela") e qual formato de cabeçalho e sequência é esperado. É semelhante ao parâmetro para `compressobj()`, mas aceita mais intervalos de valores:

- +8 to +15: O logaritmo de base dois do tamanho da janela. A entrada deve incluir um cabeçalho e uma sequência de zlib.
- 0: Determina automaticamente o tamanho da janela no cabeçalho zlib. Suportado apenas desde o zlib 1.2.3.5.
- −8 to −15: Usa o valor absoluto de *wbits* como o logaritmo do tamanho da janela. A entrada deve ser um fluxo bruto sem cabeçalho ou sequência.
- +24 para +31 = 16 + (8 para 15): Usa os 4 bits baixos do valor como logaritmo do tamanho da janela. A entrada deve incluir um cabeçalho e sequência de gzip.
- +40 a +47 = 32 + (8 a 15): Usa os 4 bits baixos do valor como logaritmo do tamanho da janela e aceita automaticamente o formato zlib ou gzip.

Ao descompactar um fluxo, o tamanho da janela não deve ser menor que o tamanho originalmente usado para compactar o fluxo; o uso de um valor muito pequeno pode resultar em uma exceção `error`. O valor padrão *wbits* corresponde ao maior tamanho da janela e requer que um cabeçalho e uma sequência de zlib sejam incluídos.

bufsize é o tamanho inicial do buffer usado para armazenar dados descompactados. Se for necessário mais espaço, o tamanho do buffer será aumentado conforme necessário, para que você não precise obter esse valor exatamente correto; sintonizando, apenas algumas chamadas serão salvas em `malloc()`.

Alterado na versão 3.6: *wbits* and *bufsize* can be used as keyword arguments.

zlib.**decompressobj**(*wbits=MAX_WBITS*[, *zdict*])

Returns a decompression object, to be used for decompressing data streams that won't fit into memory at once.

The *wbits* parameter controls the size of the history buffer (or the "window size"), and what header and trailer format is expected. It has the same meaning as *described for decompress()*.

The *zdict* parameter specifies a predefined compression dictionary. If provided, this must be the same dictionary as was used by the compressor that produced the data that is to be decompressed.

Nota: If *zdict* is a mutable object (such as a `bytearray`), you must not modify its contents between the call to `decompressobj()` and the first call to the decompressor's `decompress()` method.

Alterado na versão 3.3: Adicionou parâmetro *zdict*.

Compression objects support the following methods:

Compress.**compress**(*data*)

Compress *data*, returning a bytes object containing compressed data for at least part of the data in *data*. This data should be concatenated to the output produced by any preceding calls to the `compress()` method. Some input may be kept in internal buffers for later processing.

Compress.**flush**([*mode*])

All pending input is processed, and a bytes object containing the remaining compressed output is returned. *mode* can be selected from the constants `Z_NO_FLUSH`, `Z_PARTIAL_FLUSH`, `Z_SYNC_FLUSH`, `Z_FULL_FLUSH`, `Z_BLOCK` (zlib 1.2.3.4), or `Z_FINISH`, defaulting to `Z_FINISH`. Except `Z_FINISH`, all constants allow compressing further bytestrings of data, while `Z_FINISH` finishes the compressed stream and prevents compressing any more data. After calling `flush()` with *mode* set to `Z_FINISH`, the `compress()` method cannot be called again; the only realistic action is to delete the object.

Compress.**copy**()
Returns a copy of the compression object. This can be used to efficiently compress a set of data that share a common initial prefix.

Decompression objects support the following methods and attributes:

Decompress.**unused_data**
A bytes object which contains any bytes past the end of the compressed data. That is, this remains `b""` until the last byte that contains compression data is available. If the whole bytestring turned out to contain compressed data, this is `b""`, an empty bytes object.

Decompress.**unconsumed_tail**
A bytes object that contains any data that was not consumed by the last *decompress()* call because it exceeded the limit for the uncompressed data buffer. This data has not yet been seen by the zlib machinery, so you must feed it (possibly with further data concatenated to it) back to a subsequent *decompress()* method call in order to get correct output.

Decompress.**eof**
A boolean indicating whether the end of the compressed data stream has been reached.

This makes it possible to distinguish between a properly-formed compressed stream, and an incomplete or truncated one.

Novo na versão 3.3.

Decompress.**decompress**(*data*, *max_length=0*)
Decompress *data*, returning a bytes object containing the uncompressed data corresponding to at least part of the data in *string*. This data should be concatenated to the output produced by any preceding calls to the *decompress()* method. Some of the input data may be preserved in internal buffers for later processing.

If the optional parameter *max_length* is non-zero then the return value will be no longer than *max_length*. This may mean that not all of the compressed input can be processed; and unconsumed data will be stored in the attribute *unconsumed_tail*. This bytestring must be passed to a subsequent call to *decompress()* if decompression is to continue. If *max_length* is zero then the whole input is decompressed, and *unconsumed_tail* is empty.

Alterado na versão 3.6: *max_length* can be used as a keyword argument.

Decompress.**flush**([*length*])
All pending input is processed, and a bytes object containing the remaining uncompressed output is returned. After calling *flush()*, the *decompress()* method cannot be called again; the only realistic action is to delete the object.

The optional parameter *length* sets the initial size of the output buffer.

Decompress.**copy**()
Returns a copy of the decompression object. This can be used to save the state of the decompressor midway through the data stream in order to speed up random seeks into the stream at a future point.

Information about the version of the zlib library in use is available through the following constants:

zlib.**ZLIB_VERSION**
The version string of the zlib library that was used for building the module. This may be different from the zlib library actually used at runtime, which is available as *ZLIB_RUNTIME_VERSION*.

zlib.**ZLIB_RUNTIME_VERSION**
The version string of the zlib library actually loaded by the interpreter.

Novo na versão 3.3.

Ver também:

Module *gzip* Reading and writing **gzip**-format files.

http://www.zlib.net The zlib library home page.

http://www.zlib.net/manual.html The zlib manual explains the semantics and usage of the library's many functions.

13.2 gzip — Support for gzip files

Código Fonte: Lib/gzip.py

This module provides a simple interface to compress and decompress files just like the GNU programs **gzip** and **gunzip** would.

The data compression is provided by the *zlib* module.

The *gzip* module provides the *GzipFile* class, as well as the *open()*, *compress()* and *decompress()* convenience functions. The *GzipFile* class reads and writes **gzip**-format files, automatically compressing or decompressing the data so that it looks like an ordinary *file object*.

Note that additional file formats which can be decompressed by the **gzip** and **gunzip** programs, such as those produced by **compress** and **pack**, are not supported by this module.

Este módulo define os seguintes itens:

gzip.**open**(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)
Open a gzip-compressed file in binary or text mode, returning a *file object*.

The *filename* argument can be an actual filename (a *str* or *bytes* object), or an existing file object to read from or write to.

The *mode* argument can be any of `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, `'wb'`, `'x'` or `'xb'` for binary mode, or `'rt'`, `'at'`, `'wt'`, or `'xt'` for text mode. The default is `'rb'`.

The *compresslevel* argument is an integer from 0 to 9, as for the *GzipFile* constructor.

For binary mode, this function is equivalent to the *GzipFile* constructor: `GzipFile(filename, mode, compresslevel)`. In this case, the *encoding*, *errors* and *newline* arguments must not be provided.

For text mode, a *GzipFile* object is created, and wrapped in an *io.TextIOWrapper* instance with the specified encoding, error handling behavior, and line ending(s).

Alterado na versão 3.3: Added support for *filename* being a file object, support for text mode, and the *encoding*, *errors* and *newline* arguments.

Alterado na versão 3.4: Added support for the `'x'`, `'xb'` and `'xt'` modes.

Alterado na versão 3.6: Aceita um *path-like object*.

class gzip.**GzipFile**(*filename=None*, *mode=None*, *compresslevel=9*, *fileobj=None*, *mtime=None*)
Constructor for the *GzipFile* class, which simulates most of the methods of a *file object*, with the exception of the `truncate()` method. At least one of *fileobj* and *filename* must be given a non-trivial value.

The new class instance is based on *fileobj*, which can be a regular file, an *io.BytesIO* object, or any other object which simulates a file. It defaults to `None`, in which case *filename* is opened to provide a file object.

When *fileobj* is not `None`, the *filename* argument is only used to be included in the **gzip** file header, which may include the original filename of the uncompressed file. It defaults to the filename of *fileobj*, if discernible; otherwise, it defaults to the empty string, and in this case the original filename is not included in the header.

The *mode* argument can be any of `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, `'wb'`, `'x'`, or `'xb'`, depending on whether the file will be read or written. The default is the mode of *fileobj* if discernible; otherwise, the default is `'rb'`.

Note that the file is always opened in binary mode. To open a compressed file in text mode, use *open()* (or wrap your *GzipFile* with an *io.TextIOWrapper*).

The *compresslevel* argument is an integer from 0 to 9 controlling the level of compression; 1 is fastest and produces the least compression, and 9 is slowest and produces the most compression. 0 is no compression. The default is 9.

The *mtime* argument is an optional numeric timestamp to be written to the last modification time field in the stream when compressing. It should only be provided in compression mode. If omitted or None, the current time is used. See the *mtime* attribute for more details.

Calling a *GzipFile* object's close() method does not close *fileobj*, since you might wish to append more material after the compressed data. This also allows you to pass an *io.BytesIO* object opened for writing as *fileobj*, and retrieve the resulting memory buffer using the *io.BytesIO* object's *getvalue()* method.

GzipFile supports the *io.BufferedIOBase* interface, including iteration and the with statement. Only the truncate() method isn't implemented.

GzipFile also provides the following method and attribute:

peek(*n*)
Read *n* uncompressed bytes without advancing the file position. At most one single read on the compressed stream is done to satisfy the call. The number of bytes returned may be more or less than requested.

Nota: While calling *peek()* does not change the file position of the *GzipFile*, it may change the position of the underlying file object (e.g. if the *GzipFile* was constructed with the *fileobj* parameter).

Novo na versão 3.2.

mtime
When decompressing, the value of the last modification time field in the most recently read header may be read from this attribute, as an integer. The initial value before reading any headers is None.

All **gzip** compressed streams are required to contain this timestamp field. Some programs, such as **gunzip**, make use of the timestamp. The format is the same as the return value of *time.time()* and the *st_mtime* attribute of the object returned by *os.stat()*.

Alterado na versão 3.1: Support for the with statement was added, along with the *mtime* constructor argument and *mtime* attribute.

Alterado na versão 3.2: Support for zero-padded and unseekable files was added.

Alterado na versão 3.3: The *io.BufferedIOBase.read1()* method is now implemented.

Alterado na versão 3.4: Added support for the 'x' and 'xb' modes.

Alterado na versão 3.5: Added support for writing arbitrary *bytes-like objects*. The *read()* method now accepts an argument of None.

Alterado na versão 3.6: Aceita um *path-like object*.

gzip.**compress**(*data*, *compresslevel=9*)
Compress the *data*, returning a *bytes* object containing the compressed data. *compresslevel* has the same meaning as in the *GzipFile* constructor above.

Novo na versão 3.2.

gzip.**decompress**(*data*)
Decompress the *data*, returning a *bytes* object containing the uncompressed data.

Novo na versão 3.2.

13.2.1 Exemplos de uso

Example of how to read a compressed file:

```
import gzip
with gzip.open('/home/joe/file.txt.gz', 'rb') as f:
    file_content = f.read()
```

Example of how to create a compressed GZIP file:

```
import gzip
content = b"Lots of content here"
with gzip.open('/home/joe/file.txt.gz', 'wb') as f:
    f.write(content)
```

Example of how to GZIP compress an existing file:

```
import gzip
import shutil
with open('/home/joe/file.txt', 'rb') as f_in:
    with gzip.open('/home/joe/file.txt.gz', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)
```

Example of how to GZIP compress a binary string:

```
import gzip
s_in = b"Lots of content here"
s_out = gzip.compress(s_in)
```

Ver também:

Módulo *zlib* The basic data compression module needed to support the **gzip** file format.

13.3 bz2 — Suporte para compressão bzip2

Código Fonte: Lib/bz2.py

Este módulo fornece uma interface abrangente para compactar e descompactar dados usando o algoritmo de compactação bzip2

O módulo *bz2* contém:

- A função *open()* e a classe *BZ2File* para leitura e escrita de arquivos compactados.
- As classes *BZ2Compressor* e *BZ2Decompressor* para (des)compressão incremental.
- As funções *compress()* e *decompress()* para (des)compressão de uma só vez.

All of the classes in this module may safely be accessed from multiple threads.

13.3.1 (Des)compressão de arquivos

bz2.**open**(*filename*, *mode='r'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)
Abre um arquivo compactado com bzip2 no modo binário ou texto, retornando um *objeto arquivo*.

Assim como no construtor para `BZ2File`, o argumento *filename* pode ser um nome de arquivo real (um objeto `str` ou `bytes`), ou um objeto arquivo existente para ler ou gravar.

O argumento *mode* pode ser qualquer um de `'r'`, `'rb'`, `'w'`, `'wb'`, `'x'`, `'xb'`, `'a'` ou `'ab'` para modo binário, ou `'rt'`, `'wt'`, `'xt'` ou `'at'` para modo texto. O padrão é `'rb'`.

O argumento *compresslevel* é um inteiro de 1 a 9, como para o construtor `BZ2File`.

Para o modo binário, esta função é equivalente a `BZ2File` constructor: `BZ2File(filename, mode, compresslevel=compresslevel)`. Neste caso, os argumentos *encoding*, *errors* e *newline* não devem ser fornecidos.

Para o modo texto, um objeto `BZ2File` é criado e envolto em uma instância `io.TextIOWrapper` com a codificação especificada, comportamento de tratamento de erros e final(is) de linha.

Novo na versão 3.3.

Alterado na versão 3.4: O modo `'x'` (criação exclusiva) foi adicionado.

Alterado na versão 3.6: Aceita um *path-like object*.

class bz2.**BZ2File**(*filename*, *mode='r'*, *buffering=None*, *compresslevel=9*)
Abre um arquivo compactado com bzip2 no modo binário.

Se *filename* for um objeto `str` ou `bytes`, abra o arquivo nomeado diretamente. Caso contrário, *filename* deve ser um *objeto arquivo*, que será usado para ler ou gravar os dados compactados.

O argumento *mode* pode ser `'r'` para leitura (padrão), `'w'` para substituição, `'x'` para criação exclusiva ou `'a'` para anexar. Estes podem ser equivalentemente dados como `'rb'`, `'wb'`, `'xb'` e `'ab'` respectivamente.

Se *filename* for um objeto arquivo (ao invés de um nome de arquivo real), um modo de `'w'` não truncará o arquivo e será equivalente a `'a'`.

The *buffering* argument is ignored. Its use is deprecated.

Se *mode* for `'w'` ou `'a'`, *compresslevel* pode ser um inteiro entre `1` e `9` especificando o nível de compressão: `1` ``produz a menor compressão e ``9` (padrão) produz a maior compactação.

Se *mode* for `'r'`, o arquivo de entrada pode ser a concatenação de vários fluxos compactados.

`BZ2File` fornece todos os membros especificados pelo `io.BufferedIOBase`, exceto `detach()` e `truncate()`. Iteração e a instrução `with` são suportadas.

`BZ2File` também fornece o seguinte método:

peek([*n*])
Retorna dados armazenados em buffer sem avançar a posição do arquivo. Pelo menos um byte de dados será retornado (a menos que em EOF). O número exato de bytes retornados não é especificado.

Nota: Enquanto chamar `peek()` não altera a posição do arquivo de `BZ2File`, pode alterar a posição do objeto de arquivo subjacente (por exemplo, se o `BZ2File` foi construído passando um objeto de arquivo para *filename*).

Novo na versão 3.3.

Alterado na versão 3.1: Suporte para a instrução `with` foi adicionado.

Alterado na versão 3.3: Os métodos `fileno()`, `readable()`, `seekable()`, `writable()`, `read1()` e `readinto()` foram adicionados.

Alterado na versão 3.3: Foi adicionado suporte para *filename* ser um *objeto arquivo* em vez de um nome de arquivo real.

Alterado na versão 3.3: O modo `'a'` (anexar) foi adicionado, juntamente com suporte para leitura de arquivos multifluxo.

Alterado na versão 3.4: O modo `'x'` (criação exclusiva) foi adicionado.

Alterado na versão 3.5: O método *read()* agora aceita um argumento de `None`.

Alterado na versão 3.6: Aceita um *path-like object*.

13.3.2 (Des)compressão incremental

class bz2.**BZ2Compressor**(*compresslevel=9*)

Cria um novo objeto compressor. Este objeto pode ser usado para compactar dados de forma incremental. Para compactação única, use a função *compress()*.

compresslevel, se fornecido, deve ser um inteiro entre `1` e `9`. O padrão é `9`.

compress(*data*)

Fornece dados para o objeto compressor. Retorna um pedaço de dados compactados, se possível, ou uma string de bytes vazia, caso contrário.

Quando você terminar de fornecer dados ao compactador, chame o método *flush()* para finalizar o processo de compressão.

flush()

Finaliza o processo de compactação. Retorna os dados compactados deixados em buffers internos.

O objeto compactador não pode ser usado após a chamada deste método.

class bz2.**BZ2Decompressor**

Cria um novo objeto descompactador. Este objeto pode ser usado para descompactar dados de forma incremental. Para compactação única, use a função *decompress()*.

Nota: Esta classe não trata de forma transparente entradas contendo múltiplos fluxos compactados, ao contrário de *decompress()* e *BZ2File*. Se você precisar descompactar uma entrada multifluxo com *BZ2Decompressor*, você deve usar um novo descompactador para cada fluxo.

---

decompress(*data*, *max_length=-1*)

Descompacta dados *data* (um *objeto bytes ou similar*), retornando dados não compactados como bytes. Alguns dos *data* podem ser armazenados em buffer internamente, para uso em chamadas posteriores para *decompress()*. Os dados retornados devem ser concatenados com a saída de qualquer chamada anterior para *decompress()*.

Se *max_length* for não negativo, retornará no máximo *max_length* bytes de dados descompactados. Se este limite for atingido e mais saída puder ser produzida, o atributo *needs_input* será definido como `False`. Neste caso, a próxima chamada para *decompress()* pode fornecer *data* como `b''` para obter mais saída.

Se todos os dados de entrada foram descompactados e retornados (seja porque era menor que *max_length* bytes, ou porque *max_length* era negativo), o atributo *needs_input* será definido como `True` .

A tentativa de descompactar os dados após o final do fluxo ser atingido gera um *EOFError*. Quaisquer dados encontrados após o final do fluxo são ignorados e salvos no atributo *unused_data*.

Alterado na versão 3.5: Adicionado o parâmetro *max_length*.

eof
`True` se o marcador de fim de fluxo foi atingido.

Novo na versão 3.3.

unused_data
Dados encontrados após o término do fluxo compactado.

Se este atributo for acessado antes do final do fluxo ser alcançado, seu valor será `b''`.

needs_input
`False` se o método *decompress()* puder fornecer mais dados descompactados antes de exigir uma nova entrada descompactada.

Novo na versão 3.5.

13.3.3 (De)compressão de uma só vez (one-shot)

bz2.**compress**(*data*, *compresslevel=9*)
Compacta *data*, um objeto bytes ou similar.

compresslevel, se fornecido, deve ser um inteiro entre `1` e `9`. O padrão é `9`.

Para compressão incremental, use um *BZ2Compressor*.

bz2.**decompress**(*data*)
Descompacta *data*, um objeto bytes ou similar.

Se *data* for a concatenação de vários fluxos compactados, descompacta todos os fluxos.

Para descompressão incremental, use um *BZ2Decompressor*.

Alterado na versão 3.3: Suporte para entradas multifluxo foi adicionado.

13.3.4 Exemplos de uso

Abaixo estão alguns exemplos de uso típico do módulo *bz2*.

Usando *compress()* e *decompress()* para demonstrar a compactação de ida e volta:

```
>>> import bz2
```

```
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
```

```
>>> c = bz2.compress(data)
>>> len(data) / len(c)  # Data compression ratio
1.513595166163142
```

```
>>> d = bz2.decompress(c)
>>> data == d  # Check equality to original object after round-trip
True
```

Usando *BZ2Compressor* para compressão incremental:

```
>>> import bz2
```

```
>>> def gen_data(chunks=10, chunksize=1000):
...     """Yield incremental blocks of chunksize bytes."""
...     for _ in range(chunks):
...         yield b"z" * chunksize
...
>>> comp = bz2.BZ2Compressor()
>>> out = b""
>>> for chunk in gen_data():
...     # Provide data to the compressor object
...     out = out + comp.compress(chunk)
...
>>> # Finish the compression process.  Call this once you have
>>> # finished providing data to the compressor.
>>> out = out + comp.flush()
```

O exemplo acima usa um fluxo de dados muito "não aleatório" (um fluxo de partes *b"z"*). Dados aleatórios tendem a compactar mal, enquanto dados ordenados e repetitivos geralmente produzem uma alta taxa de compactação.

Escrevendo e lendo um arquivo compactado com bzip2 no modo binário:

```
>>> import bz2
```

```
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
```

```
>>> with bz2.open("myfile.bz2", "wb") as f:
...     # Write compressed data to file
...     unused = f.write(data)
```

```
>>> with bz2.open("myfile.bz2", "rb") as f:
...     # Decompress data from file
...     content = f.read()
```

```
>>> content == data  # Check equality to original object after round-trip
True
```

13.4 lzma — Compactação usando o algoritmo LZMA

Novo na versão 3.3.

Código Fonte: Lib/lzma.py

Este módulo fornece classes e funções de conveniência para compactar e descompactar dados usando o algoritmo de compactação LZMA. Também está incluída uma interface de arquivo que oferece suporte aos formatos de arquivo `.xz` e legado `.lzma` usados pelo utilitário **xz**, bem como fluxos brutos compactados.

The interface provided by this module is very similar to that of the *bz2* module. However, note that *LZMAFile* is *not* thread-safe, unlike *bz2.BZ2File*, so if you need to use a single *LZMAFile* instance from multiple threads, it is necessary to protect it with a lock.

exception lzma.**LZMAError**
Essa exceção é levantada quando ocorre um erro durante a compactação ou descompactação ou durante a inicialização do estado compactador/descompactador.

13.4.1 Lendo e escrevendo arquivos compactados

lzma.**open**(*filename*, *mode="rb"*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*, *encoding=None*, *errors=None*, *newline=None*)
Abre um arquivo compactado com LZMA no modo binário ou texto, retornando um *objeto arquivo*.

O argumento *filename* pode ser um nome de arquivo real (dado como um objeto *str*, *bytes* ou caminho ou similar), neste caso o arquivo nomeado é aberto , ou pode ser um objeto arquivo existente para leitura ou escrita.

O argumento *mode* pode ser qualquer um de `"r"`, `"rb"`, `"w"`, `"wb"`, `"x"`, `"xb"`, `"a"` ou `"ab"` para modo binário, ou `"rt"`, `"wt"`, `"xt"` , ou `"at"` para o modo de texto. O padrão é `"rb"`.

Ao abrir um arquivo para leitura, os argumentos *format* e *filters* têm os mesmos significados que em *LZMADecompressor*. Neste caso, os argumentos *check* e *preset* não devem ser usados.

Ao abrir um arquivo para escrita, os argumentos *format*, *check*, *preset* e *filters* têm os mesmos significados que em *LZMACompressor*.

Para o modo binário, esta função é equivalente ao construtor *LZMAFile*: `LZMAFile(filename, mode, ...)`. Nesse caso, os argumentos *encoding*, *errors* e *newline* não devem ser fornecidos.

Para o modo texto, um objeto *LZMAFile* é criado e encapsulado em uma instância *io.TextIOWrapper* com a codificação especificada, comportamento de tratamento de erros e final(is) de linha.

Alterado na versão 3.4: Adicionado suporte para os modos `"x"`, `"xb"` e `"xt"`.

Alterado na versão 3.6: Aceita um *path-like object*.

class lzma.**LZMAFile**(*filename=None*, *mode="r"*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*)
Abre um arquivo compactado com LZMA no modo binário.

Um *LZMAFile* pode envolver um *objeto arquivo* já aberto, ou operar diretamente em um arquivo nomeado. O argumento *filename* especifica o objeto arquivo a ser encapsulado ou o nome do arquivo a ser aberto (como um objeto *str*, *bytes* ou caminho ou similar). Ao agrupar um objeto de arquivo existente, o arquivo agrupado não será fechado quando o *LZMAFile* for fechado.

O argumento *mode* pode ser `"r"` para leitura (padrão), `"w"` para substituição, `"x"` para criação exclusiva ou `"a"` para anexar. Estes podem ser equivalentemente dados como `"rb"`, `"wb"`, `"xb"` e `"ab"` respectivamente.

Se *filename* for um objeto arquivo (em vez de um nome de arquivo real), um modo de `"w"` não truncará o arquivo e será equivalente a `"a"`.

Ao abrir um arquivo para leitura, o arquivo de entrada pode ser a concatenação de vários fluxos compactados separados. Estes são decodificados de forma transparente como um único fluxo lógico.

Ao abrir um arquivo para leitura, os argumentos *format* e *filters* têm os mesmos significados que em `LZMADecompressor`. Neste caso, os argumentos *check* e *preset* não devem ser usados.

Ao abrir um arquivo para escrita, os argumentos *format*, *check*, *preset* e *filters* têm os mesmos significados que em `LZMACompressor`.

`LZMAFile` oferece suporte a todos os membros especificados por `io.BufferedIOBase`, exceto `detach()` e `truncate()`. Iteração e a instrução `with` são suportadas.

O método a seguir também é fornecido:

peek(*size=-1*)
Retorna dados armazenados em buffer sem avançar a posição do arquivo. Pelo menos um byte de dados será retornado, a menos que o EOF tenha sido atingido. O número exato de bytes retornados não é especificado (o argumento *size* é ignorado).

Nota: Enquanto chamar `peek()` não altera a posição do arquivo de `LZMAFile`, pode alterar a posição do objeto arquivo subjacente (por exemplo, se o `LZMAFile` foi construído passando um objeto arquivo para *nome do arquivo*).

Alterado na versão 3.4: Adicionado suporte para os modos `"x"` e `"xb"`.

Alterado na versão 3.5: O método `read()` agora aceita um argumento de `None`.

Alterado na versão 3.6: Aceita um *path-like object*.

13.4.2 Compactando e descompactando dados na memória

class lzma.**LZMACompressor**(*format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)
Cria um objeto compactador, que pode ser usado para compactar dados de forma incremental.

Para uma maneira mais conveniente de compactar um único bloco de dados, consulte `compress()`.

O argumento *format* especifica qual formato de contêiner deve ser usado. Os valores possíveis são:

- **`FORMAT_XZ`: O formato de contêiner `.xz`.** Este é o formato padrão.
- **`FORMAT_ALONE`: O formato de contêiner legado `.lzma`.** Este formato é mais limitado que `.xz` – ele não oferece suporte a verificações de integridade ou filtros múltiplos.
- **`FORMAT_RAW`: Um fluxo de dados brutos, que não usa nenhum formato de contêiner.** Esse especificador de formato não oferece suporte a verificações de integridade e exige que você sempre especifique uma cadeia de filtros personalizada (para compactação e descompactação). Além disso, dados compactados dessa maneira não podem ser descompactados usando `FORMAT_AUTO` (veja `LZMADecompressor`).

O argumento *check* especifica o tipo de verificação de integridade a ser incluída nos dados compactados. Essa verificação é usada ao descompactar, para garantir que os dados não foram corrompidos. Os valores possíveis são:

- `CHECK_NONE`: Sem verificação de integridade. Este é o padrão (e o único valor aceitável) para `FORMAT_ALONE` e `FORMAT_RAW`.
- `CHECK_CRC32`: Verificação de redundância cíclica de 32 bits.
- `CHECK_CRC64`: Verificação de redundância cíclica de 64 bits. Este é o padrão para `FORMAT_XZ`.
- `CHECK_SHA256`: Algoritmo de hash seguro de 256 bits.

Se a verificação especificada não for suportada, uma exceção `LZMAError` será levantada.

As configurações de compactação podem ser especificadas como um nível de compactação predefinido (com o argumento *preset*) ou em detalhes como uma cadeia de filtros personalizada (com o argumento *filters*).

O argumento *preset* (se fornecido) deve ser um inteiro entre `0` e `9` (inclusive), opcionalmente com OR com a constante `PRESET_EXTREME`. Se nem *preset* nem *filters* forem fornecidos, o comportamento padrão é usar `PRESET_DEFAULT` (nível predefinido `6`). Predefinições mais altas produzem uma saída menor, mas tornam o processo de compactação mais lento.

Nota: Além de consumir mais CPU, a compactação com predefinições mais altas também requer muito mais memória (e produz uma saída que precisa de mais memória para descompactar). Com a predefinição `9` por exemplo, a sobrecarga para um objeto `LZMACompressor` pode chegar a 800 MiB. Por esse motivo, geralmente é melhor ficar com a predefinição padrão.

O argumento *filters* (se fornecido) deve ser um especificador de cadeia de filtros. Veja *Specifying custom filter chains* para detalhes.

compress(*data*)
Compress *data* (a `bytes` object), returning a `bytes` object containing compressed data for at least part of the input. Some of *data* may be buffered internally, for use in later calls to `compress()` and `flush()`. The returned data should be concatenated with the output of any previous calls to `compress()`.

flush()
Finish the compression process, returning a `bytes` object containing any data stored in the compressor's internal buffers.

The compressor cannot be used after this method has been called.

class lzma.**LZMADecompressor**(*format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)
Create a decompressor object, which can be used to decompress data incrementally.

For a more convenient way of decompressing an entire compressed stream at once, see `decompress()`.

The *format* argument specifies the container format that should be used. The default is `FORMAT_AUTO`, which can decompress both `.xz` and `.lzma` files. Other possible values are `FORMAT_XZ`, `FORMAT_ALONE`, and `FORMAT_RAW`.

The *memlimit* argument specifies a limit (in bytes) on the amount of memory that the decompressor can use. When this argument is used, decompression will fail with an `LZMAError` if it is not possible to decompress the input within the given memory limit.

The *filters* argument specifies the filter chain that was used to create the stream being decompressed. This argument is required if *format* is `FORMAT_RAW`, but should not be used for other formats. See *Specifying custom filter chains* for more information about filter chains.

Nota: This class does not transparently handle inputs containing multiple compressed streams, unlike `decompress()` and `LZMAFile`. To decompress a multi-stream input with `LZMADecompressor`, you must create a new decompressor for each stream.

---

decompress(*data*, *max_length=-1*)
Descompacta dados *data* (um *objeto bytes ou similar*), retornando dados não compactados como bytes. Alguns dos *data* podem ser armazenados em buffer internamente, para uso em chamadas posteriores para `decompress()`. Os dados retornados devem ser concatenados com a saída de qualquer chamada anterior para `decompress()`.

Se *max_length* for não negativo, retornará no máximo *max_length* bytes de dados descompactados. Se este limite for atingido e mais saída puder ser produzida, o atributo `needs_input` será definido como `False`. Neste caso, a próxima chamada para `decompress()` pode fornecer *data* como `b''` para obter mais saída.

Se todos os dados de entrada foram descompactados e retornados (seja porque era menor que *max_length* bytes, ou porque *max_length* era negativo), o atributo `needs_input` será definido como `True` .

A tentativa de descompactar os dados após o final do fluxo ser atingido gera um *EOFError*. Quaisquer dados encontrados após o final do fluxo são ignorados e salvos no atributo `unused_data`.

Alterado na versão 3.5: Adicionado o parâmetro *max_length*.

check
The ID of the integrity check used by the input stream. This may be `CHECK_UNKNOWN` until enough of the input has been decoded to determine what integrity check it uses.

eof
`True` se o marcador de fim de fluxo foi atingido.

unused_data
Dados encontrados após o término do fluxo compactado.

Before the end of the stream is reached, this will be `b""`.

needs_input
`False` se o método `decompress()` puder fornecer mais dados descompactados antes de exigir uma nova entrada descompactada.

Novo na versão 3.5.

`lzma.`**`compress`**(*data*, *format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)
Compress *data* (a `bytes` object), returning the compressed data as a `bytes` object.

See `LZMACompressor` above for a description of the *format*, *check*, *preset* and *filters* arguments.

`lzma.`**`decompress`**(*data*, *format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)
Decompress *data* (a `bytes` object), returning the uncompressed data as a `bytes` object.

If *data* is the concatenation of multiple distinct compressed streams, decompress all of these streams, and return the concatenation of the results.

See `LZMADecompressor` above for a description of the *format*, *memlimit* and *filters* arguments.

13.4.3 Diversos

`lzma.`**`is_check_supported`**(*check*)
Return `True` if the given integrity check is supported on this system.

`CHECK_NONE` and `CHECK_CRC32` are always supported. `CHECK_CRC64` and `CHECK_SHA256` may be unavailable if you are using a version of **liblzma** that was compiled with a limited feature set.

13.4.4 Specifying custom filter chains

A filter chain specifier is a sequence of dictionaries, where each dictionary contains the ID and options for a single filter. Each dictionary must contain the key `"id"`, and may contain additional keys to specify filter-dependent options. Valid filter IDs are as follows:

- **Filtro Compression:**
  - `FILTER_LZMA1` (para ser usado com `FORMAT_ALONE`)
  - `FILTER_LZMA2` (para ser utilizado com `FORMAT_XZ` and `FORMAT_RAW`)
- **Delta filter:**
  - `FILTER_DELTA`
- **Branch-Call-Jump (BCJ) filters:**
  - `FILTER_X86`
  - `FILTER_IA64`
  - `FILTER_ARM`
  - `FILTER_ARMTHUMB`
  - `FILTER_POWERPC`
  - `FILTER_SPARC`

A filter chain can consist of up to 4 filters, and cannot be empty. The last filter in the chain must be a compression filter, and any other filters must be delta or BCJ filters.

Compression filters support the following options (specified as additional entries in the dictionary representing the filter):

- `preset`: A compression preset to use as a source of default values for options that are not specified explicitly.
- `dict_size`: Dictionary size in bytes. This should be between 4 KiB and 1.5 GiB (inclusive).
- `lc`: Number of literal context bits.
- `lp`: Number of literal position bits. The sum `lc + lp` must be at most 4.
- `pb`: Number of position bits; must be at most 4.
- `mode`: `MODE_FAST` or `MODE_NORMAL`.
- `nice_len`: What should be considered a "nice length" for a match. This should be 273 or less.
- `mf`: What match finder to use – `MF_HC3`, `MF_HC4`, `MF_BT2`, `MF_BT3`, or `MF_BT4`.
- `depth`: Maximum search depth used by match finder. 0 (default) means to select automatically based on other filter options.

The delta filter stores the differences between bytes, producing more repetitive input for the compressor in certain circumstances. It supports one option, `dist`. This indicates the distance between bytes to be subtracted. The default is 1, i.e. take the differences between adjacent bytes.

The BCJ filters are intended to be applied to machine code. They convert relative branches, calls and jumps in the code to use absolute addressing, with the aim of increasing the redundancy that can be exploited by the compressor. These filters support one option, `start_offset`. This specifies the address that should be mapped to the beginning of the input data. The default is 0.

13.4.5 Exemplos

Reading in a compressed file:

```
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

Criando um arquivo comprimido:

```
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

Compressing data in memory:

```
import lzma
data_in = b"Insert Data Here"
data_out = lzma.compress(data_in)
```

Compressão incremental:

```
import lzma
lzc = lzma.LZMACompressor()
out1 = lzc.compress(b"Some data\n")
out2 = lzc.compress(b"Another piece of data\n")
out3 = lzc.compress(b"Even more data\n")
out4 = lzc.flush()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

Writing compressed data to an already-open file:

```
import lzma
with open("file.xz", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with lzma.open(f, "w") as lzf:
        lzf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

Creating a compressed file using a custom filter chain:

```
import lzma
my_filters = [
    {"id": lzma.FILTER_DELTA, "dist": 5},
    {"id": lzma.FILTER_LZMA2, "preset": 7 | lzma.PRESET_EXTREME},
]
with lzma.open("file.xz", "w", filters=my_filters) as f:
    f.write(b"blah blah blah")
```

13.5 zipfile — Trabalha com arquivos ZIP

Código Fonte: Lib/zipfile.py

O formato de arquivo ZIP é um padrão de compactação e arquivamento. Este módulo fornece ferramentas para criar, ler, escrever, adicionar, e listar um arquivo ZIP. Qualquer uso avançado deste módulo vai exigir um entendimento do formato, como definido em PKZIP Application Note.

Esse módulo atualmente não suporta arquivos ZIP multi-disco. Ele pode lidar com arquivos ZIP que usam as extensões ZIP64 (ou seja arquivos ZIP com tamanho maior do que 4 Gb). Ele suporta descriptografia de arquivos criptografados dentro do ZIP, mas atualmente não pode criar um arquivo criptografado. A descriptografia é extremamente lenta pois é implementada em Python nativo ao invés de C.

Este módulo define os seguintes itens:

exception zipfile.**BadZipFile**
Este erro ocorre para arquivos ZIP corrompidos.

Novo na versão 3.2.

exception zipfile.**BadZipfile**
Alias para *BadZipFile*, para compatibilidade com versões mais antigas de Python.

Obsoleto desde a versão 3.2.

exception zipfile.**LargeZipFile**
Este erro ocorre quando um arquivo ZIP precisa da funcionalidade ZIP64 que não está habilitada.

class zipfile.**ZipFile**
A classe para ler e escrever arquivos ZIP. Veja a seção *Objetos ZipFile* para detalhes do construtor.

class zipfile.**PyZipFile**
Classe para criar arquivos ZIP contendo bibliotecas Python.

class zipfile.**ZipInfo**(*filename='NoName'*, *date_time=(1980*, *1*, *1*, *0*, *0*, *0)*)
Classe usada para representar informação sobre um membro de um archive. Instâncias desta classe são retornadas pelos métodos *getinfo()* e *infolist()* de objetos da classe *ZipFile*. A maioria dos usuários do módulo *zipfile* não vai precisar criar, mas apenas usar objetos criados pelo módulo. *filename* deveria ser o caminho completo do membro do arquivo, e *date_time* deveria ser uma tupla contendo seis campos que descrevem o momento da última modificação no arquivo; os campos são descritos na seção *Objetos ZipInfo*.

zipfile.**is_zipfile**(*filename*)
Retorna `True` se *filename* é um arquivo ZIP válido baseado no seu magic number, caso contrário retorna `False`. *filename* pode ser um arquivo ou um objeto file-like também.

Alterado na versão 3.1: Suporte para arquivo e objetos file-like.

zipfile.**ZIP_STORED**
Código numérico para um membro de um arquivo descompactado

zipfile.**ZIP_DEFLATED**
Código numérico para o método de compactação usual. Requer o módulo *zlib*.

zipfile.**ZIP_BZIP2**
Código numérico para o método de compactação BZIP2. Requer o módulo *bz2*.

Novo na versão 3.3.

zipfile.**ZIP_LZMA**
Código numérico para o método de compactação LZMA. Requer o módulo *lzma*.

Novo na versão 3.3.

Nota: A especificação do formato ZIP incluiu suporte para compactação bzip2 desde 2001, e para compactação LZMA desde 2006. Porém, algumas ferramentas (incluindo versões mais antigas de Python) não suportam esses métodos de compactação, e podem recusar processar o arquivo ZIP como um todo, ou falhar em extrair arquivos individuais.

Ver também:

PKZIP Notas da Aplicação Documentação do formato de arquivo ZIP feita por Phil Katz, criador do formato e dos algoritmos usados.

Info-ZIP Home Page Informações sobre o programas de arquivamento e desenvolvimento de bibliotecas do projeto Info-ZIP.

13.5.1 Objetos ZipFile

class zipfile.**ZipFile**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *compresslevel=None*)

Abre um arquivo ZIP, onde *file* pode ser um caminho para um arquivo (uma string), um objeto file-like, ou um objeto path-like.

O parâmetro *mode* deve ser `'r'` para ler um arquivo existente, `'w'` para truncar e gravar um novo arquivo, `'a'` para adicionar a um arquivo existente, ou `'x'` exclusivamente para criar e gravar um novo arquivo. Se o *mode* é `'x'` e *file* se refere a um arquivo existente, um *FileExistsError* vai ser lançado. Se o *mode* é `'a'` e *file* se refere a um arquivo ZIP existente, então arquivos adicionais são adicionados ao mesmo. Se *file* não se refere a um arquivo ZIP, então um novo arquivo ZIP é adicionado ao arquivo. Isso diz respeitoa adicionar um arquivo ZIP a um outro arquivo (como por exemplo `python.exe`). Se o *mode* é `'a'` e o arquivo não existe, ele será criado. Se o *mode* é `'r'` ou `'a'`, o arquivo deve ser percorrível.

compression é o método de compactação ZIP para usar ao escrever o arquivo, e deve ser *ZIP_STORED*, *ZIP_DEFLATED*, *ZIP_BZIP2* ou *ZIP_LZMA*; valores desconhecidos devem causar *NotImplementedError*. Se *ZIP_DEFLATED*, *ZIP_BZIP2* or *ZIP_LZMA* for especificado mas o módulo correspondente (*zlib*, *bz2* or *lzma*) não estiver disponível, é lançado um *RuntimeError*. O valor padrão é *ZIP_STORED*.

Se *allowZip64* é `True` (valor padrão), então zipfile vai criar arquivos ZIP que usem as extensões ZIP64 quando o arquivo ZIP é maior do que 4 Gb. Se é `false` *zipfile* lança uma exceção quando o arquivo ZIP precisaria das extensões ZIP64.

O parâmetro *compresslevel* controla o nível de compactação para usar ao gravar no arquivo ZIP. Quando usado *ZIP_STORED* ou *ZIP_LZMA* não tem efeito. Quando usado *ZIP_DEFLATED* inteiros de `0` a `9` são aceitos (veja *zlib* para mais informações). Quando usado *ZIP_BZIP2* inteiros de `1` a `9` são aceitos (veja *bz2* para mais informações).

Se o arquivo é criado com modo `'w'`, `'x'` ou `'a'` e então `closed()` sem adicionar nada ao arquivo, a estrutura própria para um arquivo vazio será escrita no arquivo.

ZipFile também é um gerenciador de contexto e portanto suporta a declaração `with`. Neste exemplo, *myzip* é fechado ao final da execução da declaração `with` – mesmo que ocorra uma exceção.

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

Novo na versão 3.2: Adicionado o uso de *ZipFile* como um gerenciador de contexto.

Alterado na versão 3.3: Adicionado suporte para compactação `bzip2 ` ` e :mod:`lzma`.

Alterado na versão 3.4: Extensões ZIP64 são habilitadas por padrão.

Alterado na versão 3.5: Adicionado suporte para escrever em streams não percorríveis. Adicionado suporte ao modo `'x'`.

Alterado na versão 3.6: Anteriormente, um simples *RuntimeError* era lançado para valores de compactação desconhecidos.

Alterado na versão 3.6.2: O parâmetro *file* aceita um objeto path-like.

Alterado na versão 3.7: Adicionado o parâmetro *compresslevel*.

ZipFile.**close**()
Fecha o arquivo. Você deve chamar *close()* antes de sair do seu programa ou registros essenciais não serão gravados.

ZipFile.**getinfo**(*name*)
Retorna um objeto *ZipInfo* com informações sobre o *name* do membro do arquivo. Chamar *getinfo()* para um nome não encontrado no arquivo lança um *KeyError*.

ZipFile.**infolist**()
Retorna uma lista contendo um objeto *ZipInfo* para cada membro do arquivo. Os objetos estão na mesma ordem das entradas no arquivo ZIP em disco se um arquivo existente foi aberto.

ZipFile.**namelist**()
Retorna uma lista de membros do arquivo por nome.

ZipFile.**open**(*name*, *mode='r'*, *pwd=None*, ***, *force_zip64=False*)
Acessa um membro do arquivo como um objeto file-like ou arquivo binário. *name* pode ser o nome de um arquivo membro ou um objeto *ZipInfo*. O parâmetro *mode*, se informado, deve ser `'r'` (valor padrão) or `'w'`. *pwd* é a senha usada para descriptografar arquivos ZIP criptografados.

open() também é um gerenciador de contexto e portanto suporta declarações com `with`

```
with ZipFile('spam.zip') as myzip:
    with myzip.open('eggs.txt') as myfile:
        print(myfile.read())
```

Com *mode* `'r'` o objeto file-like (`ZipExtFile`) é read-only e fornece os seguintes métodos: *read()*, *readline()*, *readlines()*, *seek()*, *tell()*, __iter__(), *__next__()*. Esses objetos podem operar independentemente do ZipFile.

Com `mode='w'`, é retornado um handle de arquivo, que suporta o método *write()*. Quando um handle de arquivo modificável é aberto, tentativas de ler ou gravar outros arquivos no arquivo ZIP lança um *ValueError*.

Ao gravar um arquivo, se o tamanho do arquivo não é conhecido mas pode exceder 2 Gb, passe `force_zip64=True` para assegurar que o formato do header é capaz de suportar arquivos grandes. Se o tamanho do arquivo é conhecido, construa um objeto *ZipInfo* com *file_size* informado, então use-o como parâmetro *name*.

Nota: Os métodos *open()*, *read()* e *extract()* podem receber um nome de arquivo ou um objeto *ZipInfo*. Voce vai gostar disso quando tentar ler um arquivo ZIP que contém membros com nomes duplicados.

Alterado na versão 3.6: Removido suporte ao `mode='U'`. Uso de *io.TextIOWrapper* para leitura de arquivos texto compactados em modo *universal newlines*.

Alterado na versão 3.6: *open()* agora pode ser usado para gravar arquivos com a opção `mode='w'`.

Alterado na versão 3.6: Chamar *open()* em um ZipFile fechado lança um *ValueError*. Anteriormente, um *RuntimeError* era lançado.

ZipFile.**extract**(*member*, *path=None*, *pwd=None*)
Extrai um membro do arquivo para o diretório atual; *member* deve ser o nome complelto ou um objeto *ZipInfo*. A informação do arquivo é extraída com maior precisão possível. *path* especifica um outro diretório em que deve ser gravado. *member* pode ser um nome de arquivo ou um objeto *ZipInfo*. *pwd* é a senha usada para criptografar arquivos.

Retorna o caminho normalizado criado (um diretório ou novo arquivo).

Nota: Se um nome de arquivo membro é um caminho absoluto, o drive/UNC e (contra)barras no início serão removidos, por exemplo: `///foo/bar` se torna `foo/bar` no Unix, e `C:\foo\bar` vira `foo\bar` no Windows. E todos os componentes `"..."` no nome de um arquivo membro serão removidos, por exemplo: `../../foo../../ba..r` vira `foo../ba..r`. No Windows caracteres ilegais (`:`, `<`, `>`, `|`, `"`, `?`, and `*`) são substituídos por underscore (_).

Alterado na versão 3.6: Chamar *extract()* em um ZipFile fechado lança um *ValueError*. Anteriormente, um *RuntimeError* era lançado.

Alterado na versão 3.6.2: O parâmetro *path* aceita um objeto :term:‘path-like.

ZipFile.**extractall**(*path=None*, *members=None*, *pwd=None*)
Extrai todos os membros de um arquivo para o diretório atual. *path* especifica um diretório diferente para gravar os arquivos extraídos. *members* é opcional e deve ser um sub-conjunto da lista retornada por *namelist()*. *pwd* é uma senha usada para criptografar arquivos.

Aviso: Nunca extraia arquivos de fontes não confiáveis sem inspeção prévia. É possível que os arquivos sejam criados fora do *path*, por exemplo membros que tem nomes absolutos de arquivos começando com `"/"` ou nomes com dois pontos `".."`. Este módulo tenta prevenir isto. Veja nota em *extract()*.

Alterado na versão 3.6: Chamar *extractall()* em um ZipFile fechado lança um *ValueError*. Anteriormente, um *RuntimeError* era lançado.

Alterado na versão 3.6.2: O parâmetro *path* aceita um objeto :term:‘path-like.

ZipFile.**printdir**()
Imprime a tabela de conteúdos de um arquivo para `sys.stdout`.

ZipFile.**setpassword**(*pwd*)
Seta *pwd* como senha padrão para extrair arquivos criptografados.

ZipFile.**read**(*name*, *pwd=None*)
Retorna os bytes do arquivo *name* no arquivo compactado. *name* é o nome do arquivo no arquivo compactado, ou um objeto *ZipInfo*. O arquivo compactado deve estar aberto para leitura ou append. *pwd* é a senha usada para criptografar arquivos e, se especificada, vai sobrepor a senha padrão configurada com *setpassword()*. Chamar *read()* em um ZipFile que use um método de compactação diferente de *ZIP_STORED*, *ZIP_DEFLATED*, *ZIP_BZIP2* ou *ZIP_LZMA* lança um *NotImplementedError*. Um erro também é lançado se o módulo de compactação correspondente não está disponível.

Alterado na versão 3.6: Chamar *read()* em um ZipFile fechado lança um *ValueError*. Anteriormente, um *RuntimeError* era lançado.

ZipFile.**testzip**()
Lê todos os arquivos no arquivo compactado e verifica seus CRC's e cabeçalhos de arquivo. Retorna o nome do primeiro arquivo corrompido, or então retorna `None`.

Alterado na versão 3.6: Chamar *testzip()* em um ZipFile fechado lança um *ValueError*. Anteriormente, um *RuntimeError* era lançado.

ZipFile.**write**(*filename*, *arcname=None*, *compress_type=None*, *compresslevel=None*)
Grava o arquivo chamado *filename* no arquivo compactado, dando ao arquivo compactado o nome *arcname* (por padrão, este é o mesmo de *filename*, mas sem a letra do drive e com separadores removidos do início do nome). Se informado, *compress_type* sobrescreve o valor dado ao parâmetro *compression* do construtor para a nova entrada. Da mesma forma, *compresslevel* vai sobrescrever o construtor se informado. O arquivo compactado deve ser aberto em modo `'w'`, `'x'` ou `'a'`.

Nota: Nomes de arquivo compactado devem ser relativos a raiz do mesmo, isto é, não devem começar com um separador de caminho.

Nota: Se `arcname` (ou `filename`, se `arcname` não for informado) contém um byte nulo, o nome do arquivo no arquivo compactado será truncado no byte nulo.

Alterado na versão 3.6: Chamar *write()* em um ZipFile criado com modo `'r'` ou em um ZipFile fechado lança um *ValueError*. Anteriormente, um *RuntimeError* era lançado.

ZipFile.**writestr**(*zinfo_or_arcname*, *data*, *compress_type=None*, *compresslevel=None*)
Grava um arquivo no arquivo compactado. O conteúdo é *data*, que pode ser uma instância de *str* ou de *bytes*; Se é uma *str*, ela é encodada como UTF-8 primeiro. *zinfo_or_arcname* é o nome que será dado ao arquivo no arquivo compactado, ou uma instância de *ZipInfo*. Se é uma instância, pelo menos o nome do arquivo, a data, e a hora devem ser informados. Se é um nome, a data e hora recebem a data e hora atual. O arquivo compactado deve ser aberto em modo `'w'`, `'x'` ou `'a'`.

Se informado, *compress_type* sobrescreve o valor do parâmetro *compression* do construtor para a nova entrada, ou no *zinfo_or_arcname* (se é uma instância de *ZipInfo*). Da mesma forma, *compresslevel* vai sobrescrever o construtor se informado.

Nota: Quando é passada uma instância de *ZipInfo* ou o parâmetro *zinfo_or_arcname*, o método de compactação usado será aquele especificado no *compress_type* da instância de *ZipInfo*. Por padrão, o construtor da classe *ZipInfo* seta este membro para *ZIP_STORED*.

Alterado na versão 3.2: O argumento *compress_type*.

Alterado na versão 3.6: Chamar *writestr()* em um ZipFile criado com modo `'r'` ou em um ZipFile fechado lança um *ValueError*. Anteriormente, um *RuntimeError* era lançado.

Os seguintes atributos de dados também estão disponíveis:

ZipFile.**filename**
Nome do arquivo ZIP.

ZipFile.**debug**
O nível de saída de debug para usar. Pode ser setado de *0*‘ (valor padrão, sem nenhuma saída) a `3` (com mais saída). A informação de debug é escrita em `sys.stdout`.

ZipFile.**comment**
O comentário associado ao arquivo ZIP como um objeto :class *bytes*. Se atribuir um comentário a uma instância *ZipFile* criada com o modo `'w'`, `'x'` ou `'a'`, não deve ser maior que 65535 bytes. Comentários mais longos do que isso serão truncados.

13.5.2 Objetos PyZipFile

O construtor *PyZipFile* usa os mesmos parâmetros que o construtor *ZipFile*, e um parâmetro adicional, *otimize*.

class zipfile.**PyZipFile**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *optimize=-1*)

Novo na versão 3.2: O parâmetro *optimize*.

Alterado na versão 3.4: Extensões ZIP64 são habilitadas por padrão.

As instâncias têm um método além daqueles dos objetos *ZipFile*:

writepy(*pathname*, *basename=''*, *filterfunc=None*)

Pesquisa por arquivos `*.py` e adiciona o arquivo correspondente ao arquivo.

Se o parâmetro *optimize* para *PyZipFile* não foi fornecido ou `-1`, o arquivo correspondente é um arquivo `*.pyc`, compilando se necessário.

Se o parâmetro *Optimize* para *PyZipFile* era `0`, `1` ou `2`, apenas arquivos com esse nível de otimização (ver *compile()*) são adicionados ao o arquivo, compilando se necessário.

Se *pathname* for um arquivo, o nome do arquivo deverá terminar com `.py`, e apenas o arquivo (`*.pyc` correspondente) será adicionado no nível superior (sem informações do caminho). Se *pathname* for um arquivo que não termine com `.py`, um *RuntimeError* será levantado. Se for um diretório, e o diretório não for um diretório de pacotes, todos os arquivos `*.pyc` serão adicionados no nível superior. Se o diretório for um diretório de pacotes, todos `*.pyc` serão adicionados sob o nome do pacote como um caminho de arquivo e, se algum subdiretório for um diretório de pacotes, todos serão adicionados recursivamente na ordem de classificação.

basename destina-se apenas a uso interno.

filterfunc, se fornecido, deve ser uma função que recebe um único argumento de string. Cada caminho será passado (incluindo cada caminho de arquivo completo individual) antes de ser adicionado ao arquivo. Se *filterfunc* retornar um valor falso, o caminho não será adicionado e, se for um diretório, seu conteúdo será ignorado. Por exemplo, se nossos arquivos de teste estão todos nos diretórios `test` ou começam com a string `test_`, podemos usar um *filterfunc* para excluí-los:

```
>>> zf = PyZipFile('myprog.zip')
>>> def notests(s):
...     fn = os.path.basename(s)
...     return (not (fn == 'test' or fn.startswith('test_')))
>>> zf.writepy('myprog', filterfunc=notests)
```

O método *writepy()* faz arquivos com nomes de arquivo como este:

```
string.pyc                      # Top level name
test/__init__.pyc               # Package directory
test/testall.pyc                # Module test.testall
test/bogus/__init__.pyc         # Subpackage directory
test/bogus/myfile.pyc           # Submodule test.bogus.myfile
```

Novo na versão 3.4: O parâmetro *filterfunc*.

Alterado na versão 3.6.2: O parâmetro *pathname* aceita um *objeto caminho ou similar*.

Alterado na versão 3.7: A recursão classifica as entradas de diretório.

13.5.3 Objetos ZipInfo

Instâncias da classe *ZipInfo* são retornadas pelos métodos *getinfo()* e *infolist()* dos objetos *ZipFile*. Cada objeto armazena informações sobre um único membro do arquivo ZIP.

Existe um método de classe para fazer uma instância *ZipInfo* para um arquivo de sistema de arquivos:

classmethod ZipInfo.**from_file**(*filename*, *arcname=None*)
Constrói uma instância *ZipInfo* para um arquivo no sistema de arquivos, em preparação para adicioná-lo a um arquivo zip.

filename deve ser o caminho para um arquivo ou diretório no sistema de arquivos.

Se *arcname* for especificado, ele será usado como o nome dentro do arquivo. Se *arcname* não for especificado, o nome será igual a *filename*, mas com qualquer letra de unidade e separadores de caminho removidos.

Novo na versão 3.6.

Alterado na versão 3.6.2: O parâmetro *filename* aceita um *objeto caminho ou similar*.

As instâncias têm os seguintes métodos e atributos:

ZipInfo.**is_dir**()
Retorna `True` se este membro do arquivo for um diretório.

Isso usa o nome da entrada: os diretórios devem sempre terminar com `/`.

Novo na versão 3.6.

ZipInfo.**filename**
Nome do arquivo no pacote.

ZipInfo.**date_time**
A hora e a data da última modificação do membro do arquivo. Esta é uma tupla de seis valores:

Index	Valor
0	Ano (>= 1980)
1	Mês (iniciado em 1)
2	Dia do mês (iniciado em 1)
3	Horas (iniciado em 0)
4	Minutos (base zero)
5	Segundos (iniciado em 0)

Nota: O formato de arquivo ZIP não oferece suporte a carimbos de data/hora anteriores a 1980.

ZipInfo.**compress_type**
Tipo de compressão do membro do pacote.

ZipInfo.**comment**
Comentário para o membro individual do pacote como um objeto *bytes*.

ZipInfo.**extra**
Dados do campo de expansão. O PKZIP Application Note contém alguns comentários sobre a estrutura interna dos dados contidos neste objeto *bytes*.

ZipInfo.**create_system**
O sistema que criou o pacote ZIP.

ZipInfo.**create_version**
A versão do PKZIP que criou o pacote ZIP.

ZipInfo.**extract_version**
A versão do PKZIP necessária para extrair o pacote.

ZipInfo.**reserved**
Deve ser zero

ZipInfo.**flag_bits**
Bits de sinalizador do ZIP.

ZipInfo.**volume**
Número de volume do cabeçalho do arquivo.

ZipInfo.**internal_attr**
Atributos internos.

ZipInfo.**external_attr**
Atributos de arquivo externo.

ZipInfo.**header_offset**
Deslocamento de byte para o cabeçalho do arquivo.

ZipInfo.**CRC**
CRC-32 do arquivo não comprimido.

ZipInfo.**compress_size**
Tamanho dos dados comprimidos.

ZipInfo.**file_size**
Tamanho do arquivo não comprimido.

13.5.4 Interface de Linha de Comando

O módulo *zipfile* fornece uma interface de linha de comando simples para interagir com arquivos ZIP.

Se você deseja criar um novo arquivo ZIP, especifique seu nome após a opção *-c* e, em seguida, liste os nomes dos arquivos que devem ser incluídos:

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

Passar um diretório também é aceitável:

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

Se você deseja extrair um arquivo ZIP para o diretório especificado, use a opção *-e*:

```
$ python -m zipfile -e monty.zip target-dir/
```

Para obter uma lista dos arquivos em um arquivo ZIP, use a opção *-l*:

```
$ python -m zipfile -l monty.zip
```

Opções de linha de comando

-l <zipfile>
--list <zipfile>
Lista arquivos em um arquivo zip.

-c <zipfile> <source1> ... <sourceN>
--create <zipfile> <source1> ... <sourceN>
Cria um arquivo zip a partir dos arquivos fonte.

-e <zipfile> <output_dir>
--extract <zipfile> <output_dir>
Extrai um arquivo zip para um diretório de destino.

-t <zipfile>
--test <zipfile>
Testa se o arquivo zip é válido ou não.

13.6 `tarfile` — Read and write tar archive files

Código Fonte: Lib/tarfile.py

The *tarfile* module makes it possible to read and write tar archives, including those using gzip, bz2 and lzma compression. Use the *zipfile* module to read or write `.zip` files, or the higher-level functions in *shutil*.

Some facts and figures:

- reads and writes *gzip*, *bz2* and *lzma* compressed archives if the respective modules are available.
- read/write support for the POSIX.1-1988 (ustar) format.
- read/write support for the GNU tar format including *longname* and *longlink* extensions, read-only support for all variants of the *sparse* extension including restoration of sparse files.
- read/write support for the POSIX.1-2001 (pax) format.
- handles directories, regular files, hardlinks, symbolic links, fifos, character devices and block devices and is able to acquire and restore file information like timestamp, access permissions and owner.

Alterado na versão 3.3: Adiciona suporte para *lzma* compression.

tarfile.**open**(*name=None*, *mode='r'*, *fileobj=None*, *bufsize=10240*, ***kwargs*)
Return a *TarFile* object for the pathname *name*. For detailed information on *TarFile* objects and the keyword arguments that are allowed, see *TarFile Objects*.

mode has to be a string of the form `'filemode[:compression]'`, it defaults to `'r'`. Here is a full list of mode combinations:

modo	action
`'r' or 'r:*'`	Open for reading with transparent compression (recommended).
`'r:'`	Open for reading exclusively without compression.
`'r:gz'`	Open for reading with gzip compression.
`'r:bz2'`	Open for reading with bzip2 compression.
`'r:xz'`	Open for reading with lzma compression.
`'x' or 'x:'`	Create a tarfile exclusively without compression. Raise an *FileExistsError* exception if it already exists.
`'x:gz'`	Create a tarfile with gzip compression. Raise an *FileExistsError* exception if it already exists.
`'x:bz2'`	Create a tarfile with bzip2 compression. Raise an *FileExistsError* exception if it already exists.
`'x:xz'`	Create a tarfile with lzma compression. Raise an *FileExistsError* exception if it already exists.
`'a' or 'a:'`	Open for appending with no compression. The file is created if it does not exist.
`'w' or 'w:'`	Open for uncompressed writing.
`'w:gz'`	Open for gzip compressed writing.
`'w:bz2'`	Open for bzip2 compressed writing.
`'w:xz'`	Open for lzma compressed writing.

Note that `'a:gz'`, `'a:bz2'` or `'a:xz'` is not possible. If *mode* is not suitable to open a certain (compressed) file for reading, *ReadError* is raised. Use *mode* `'r'` to avoid this. If a compression method is not supported, *CompressionError* is raised.

If *fileobj* is specified, it is used as an alternative to a *file object* opened in binary mode for *name*. It is supposed to be at position 0.

For modes `'w:gz'`, `'r:gz'`, `'w:bz2'`, `'r:bz2'`, `'x:gz'`, `'x:bz2'`, *tarfile.open()* accepts the keyword argument *compresslevel* (default `9`) to specify the compression level of the file.

For special purposes, there is a second format for *mode*: `'filemode|[compression]'`. *tarfile.open()* will return a *TarFile* object that processes its data as a stream of blocks. No random seeking will be done on the file. If given, *fileobj* may be any object that has a `read()` or `write()` method (depending on the *mode*). *bufsize* specifies the blocksize and defaults to `20 * 512` bytes. Use this variant in combination with e.g. `sys.stdin`, a socket *file object* or a tape device. However, such a *TarFile* object is limited in that it does not allow random access, see *Exemplos*. The currently possible modes:

Modo	Action
`'r|*'`	Open a *stream* of tar blocks for reading with transparent compression.
`'r|'`	Open a *stream* of uncompressed tar blocks for reading.
`'r|gz'`	Open a gzip compressed *stream* for reading.
`'r|bz2'`	Open a bzip2 compressed *stream* for reading.
`'r|xz'`	Open an lzma compressed *stream* for reading.
`'w|'`	Open an uncompressed *stream* for writing.
`'w|gz'`	Open a gzip compressed *stream* for writing.
`'w|bz2'`	Open a bzip2 compressed *stream* for writing.
`'w|xz'`	Open an lzma compressed *stream* for writing.

Alterado na versão 3.5: O modo `'x'` (criação exclusiva) foi adicionado.

Alterado na versão 3.6: The *name* parameter accepts a *path-like object*.

class tarfile.**TarFile**
Class for reading and writing tar archives. Do not use this class directly: use *tarfile.open()* instead. See *TarFile Objects*.

tarfile.**is_tarfile**(*name*)
Return *True* if *name* is a tar archive file, that the *tarfile* module can read.

The *tarfile* module defines the following exceptions:

exception tarfile.**TarError**
Base class for all *tarfile* exceptions.

exception tarfile.**ReadError**
Is raised when a tar archive is opened, that either cannot be handled by the *tarfile* module or is somehow invalid.

exception tarfile.**CompressionError**
Is raised when a compression method is not supported or when the data cannot be decoded properly.

exception tarfile.**StreamError**
Is raised for the limitations that are typical for stream-like *TarFile* objects.

exception tarfile.**ExtractError**
Is raised for *non-fatal* errors when using *TarFile.extract()*, but only if `TarFile.errorlevel== 2`.

exception tarfile.**HeaderError**
Is raised by *TarInfo.frombuf()* if the buffer it gets is invalid.

The following constants are available at the module level:

tarfile.**ENCODING**
The default character encoding: `'utf-8'` on Windows, the value returned by *sys.getfilesystemencoding()* otherwise.

Each of the following constants defines a tar archive format that the *tarfile* module is able to create. See section *Supported tar formats* for details.

tarfile.**USTAR_FORMAT**
POSIX.1-1988 (ustar) format.

tarfile.**GNU_FORMAT**
Formato tar GNU

tarfile.**PAX_FORMAT**
POSIX.1-2001 (pax) format.

tarfile.**DEFAULT_FORMAT**
The default format for creating archives. This is currently *GNU_FORMAT*.

Ver também:

Module *zipfile* Documentation of the *zipfile* standard module.

Operações de arquivamento Documentation of the higher-level archiving facilities provided by the standard *shutil* module.

GNU tar manual, Basic Tar Format Documentation for tar archive files, including GNU tar extensions.

13.6.1 TarFile Objects

The *TarFile* object provides an interface to a tar archive. A tar archive is a sequence of blocks. An archive member (a stored file) is made up of a header block followed by data blocks. It is possible to store a file in a tar archive several times. Each archive member is represented by a *TarInfo* object, see *TarInfo Objects* for details.

A *TarFile* object can be used as a context manager in a `with` statement. It will automatically be closed when the block is completed. Please note that in the event of an exception an archive opened for writing will not be finalized; only the internally used file object will be closed. See the *Exemplos* section for a use case.

Novo na versão 3.2: Added support for the context management protocol.

class tarfile.**TarFile**(*name=None*, *mode='r'*, *fileobj=None*, *format=DEFAULT_FORMAT*, *tarinfo=TarInfo*, *dereference=False*, *ignore_zeros=False*, *encoding=ENCODING*, *errors='surrogateescape'*, *pax_headers=None*, *debug=0*, *errorlevel=1*)

All following arguments are optional and can be accessed as instance attributes as well.

name is the pathname of the archive. *name* may be a *path-like object*. It can be omitted if *fileobj* is given. In this case, the file object's `name` attribute is used if it exists.

mode is either `'r'` to read from an existing archive, `'a'` to append data to an existing file, `'w'` to create a new file overwriting an existing one, or `'x'` to create a new file only if it does not already exist.

If *fileobj* is given, it is used for reading or writing data. If it can be determined, *mode* is overridden by *fileobj*'s mode. *fileobj* will be used from position 0.

Nota: *fileobj* is not closed, when *TarFile* is closed.

---

format controls the archive format. It must be one of the constants *USTAR_FORMAT*, *GNU_FORMAT* or *PAX_FORMAT* that are defined at module level.

The *tarinfo* argument can be used to replace the default *TarInfo* class with a different one.

If *dereference* is *False*, add symbolic and hard links to the archive. If it is *True*, add the content of the target files to the archive. This has no effect on systems that do not support symbolic links.

If *ignore_zeros* is *False*, treat an empty block as the end of the archive. If it is *True*, skip empty (and invalid) blocks and try to get as many members as possible. This is only useful for reading concatenated or damaged archives.

debug can be set from `0` (no debug messages) up to `3` (all debug messages). The messages are written to `sys.stderr`.

If *errorlevel* is `0`, all errors are ignored when using *TarFile.extract()*. Nevertheless, they appear as error messages in the debug output, when debugging is enabled. If `1`, all *fatal* errors are raised as *OSError* exceptions. If `2`, all *non-fatal* errors are raised as *TarError* exceptions as well.

The *encoding* and *errors* arguments define the character encoding to be used for reading or writing the archive and how conversion errors are going to be handled. The default settings will work for most users. See section *Unicode issues* for in-depth information.

The *pax_headers* argument is an optional dictionary of strings which will be added as a pax global header if *format* is *PAX_FORMAT*.

Alterado na versão 3.2: Use `'surrogateescape'` as the default for the *errors* argument.

Alterado na versão 3.5: O modo `'x'` (criação exclusiva) foi adicionado.

Alterado na versão 3.6: The *name* parameter accepts a *path-like object*.

classmethod TarFile.**open**(...)
Alternative constructor. The *tarfile.open()* function is actually a shortcut to this classmethod.

TarFile.**getmember**(*name*)
Return a *TarInfo* object for member *name*. If *name* can not be found in the archive, *KeyError* is raised.

Nota: If a member occurs more than once in the archive, its last occurrence is assumed to be the most up-to-date version.

TarFile.**getmembers**()
Retorna os membros do arquivo como uma lista de objetos *TarInfo*. A lista tem a mesma ordem que os membros no arquivo.

TarFile.**getnames**()
Return the members as a list of their names. It has the same order as the list returned by *getmembers()*.

TarFile.**list**(*verbose=True*, *, *members=None*)
Print a table of contents to `sys.stdout`. If *verbose* is *False*, only the names of the members are printed. If it is *True*, output similar to that of **`ls -l`** is produced. If optional *members* is given, it must be a subset of the list returned by *getmembers()*.

Alterado na versão 3.5: Added the *members* parameter.

TarFile.**next**()
Return the next member of the archive as a *TarInfo* object, when *TarFile* is opened for reading. Return *None* if there is no more available.

TarFile.**extractall**(*path="."*, *members=None*, *, *numeric_owner=False*)
Extract all members from the archive to the current working directory or directory *path*. If optional *members* is given, it must be a subset of the list returned by *getmembers()*. Directory information like owner, modification time and permissions are set after all members have been extracted. This is done to work around two problems: A directory's modification time is reset each time a file is created in it. And, if a directory's permissions do not allow writing, extracting files to it will fail.

If *numeric_owner* is *True*, the uid and gid numbers from the tarfile are used to set the owner/group for the extracted files. Otherwise, the named values from the tarfile are used.

Aviso: Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of *path*, e.g. members that have absolute filenames starting with `"/"` or filenames with two dots `".."`.

Alterado na versão 3.5: Added the *numeric_owner* parameter.

Alterado na versão 3.6: O parâmetro *path* aceita um objeto :term:'path-like.

TarFile.**extract**(*member*, *path=""*, *set_attrs=True*, *, *numeric_owner=False*)
Extract a member from the archive to the current working directory, using its full name. Its file information is extracted as accurately as possible. *member* may be a filename or a *TarInfo* object. You can specify a different directory using *path*. *path* may be a *path-like object*. File attributes (owner, mtime, mode) are set unless *set_attrs* is false.

If *numeric_owner* is *True*, the uid and gid numbers from the tarfile are used to set the owner/group for the extracted files. Otherwise, the named values from the tarfile are used.

Nota: The *extract()* method does not take care of several extraction issues. In most cases you should consider

using the `extractall()` method.

Aviso: See the warning for `extractall()`.

Alterado na versão 3.2: Added the *set_attrs* parameter.

Alterado na versão 3.5: Added the *numeric_owner* parameter.

Alterado na versão 3.6: O parâmetro *path* aceita um objeto :term:'path-like.

`TarFile.`**`extractfile`**(*member*)
Extract a member from the archive as a file object. *member* may be a filename or a `TarInfo` object. If *member* is a regular file or a link, an `io.BufferedReader` object is returned. Otherwise, `None` is returned.

Alterado na versão 3.3: Return an `io.BufferedReader` object.

`TarFile.`**`add`**(*name*, *arcname=None*, *recursive=True*, *, *filter=None*)
Add the file *name* to the archive. *name* may be any type of file (directory, fifo, symbolic link, etc.). If given, *arcname* specifies an alternative name for the file in the archive. Directories are added recursively by default. This can be avoided by setting *recursive* to `False`. Recursion adds entries in sorted order. If *filter* is given, it should be a function that takes a `TarInfo` object argument and returns the changed `TarInfo` object. If it instead returns `None` the `TarInfo` object will be excluded from the archive. See *Exemplos* for an example.

Alterado na versão 3.2: Added the *filter* parameter.

Alterado na versão 3.7: Recursion adds entries in sorted order.

`TarFile.`**`addfile`**(*tarinfo*, *fileobj=None*)
Add the `TarInfo` object *tarinfo* to the archive. If *fileobj* is given, it should be a *binary file*, and `tarinfo.size` bytes are read from it and added to the archive. You can create `TarInfo` objects directly, or by using `gettarinfo()`.

`TarFile.`**`gettarinfo`**(*name=None*, *arcname=None*, *fileobj=None*)
Create a `TarInfo` object from the result of `os.stat()` or equivalent on an existing file. The file is either named by *name*, or specified as a *file object* *fileobj* with a file descriptor. *name* may be a *path-like object*. If given, *arcname* specifies an alternative name for the file in the archive, otherwise, the name is taken from *fileobj*'s `name` attribute, or the *name* argument. The name should be a text string.

You can modify some of the `TarInfo`'s attributes before you add it using `addfile()`. If the file object is not an ordinary file object positioned at the beginning of the file, attributes such as `size` may need modifying. This is the case for objects such as `GzipFile`. The `name` may also be modified, in which case *arcname* could be a dummy string.

Alterado na versão 3.6: The *name* parameter accepts a *path-like object*.

`TarFile.`**`close`**()
Close the `TarFile`. In write mode, two finishing zero blocks are appended to the archive.

`TarFile.`**`pax_headers`**
A dictionary containing key-value pairs of pax global headers.

13.6.2 TarInfo Objects

A *TarInfo* object represents one member in a *TarFile*. Aside from storing all required attributes of a file (like file type, size, time, permissions, owner etc.), it provides some useful methods to determine its type. It does *not* contain the file's data itself.

TarInfo objects are returned by *TarFile*'s methods `getmember()`, `getmembers()` and `gettarinfo()`.

class `tarfile.`**TarInfo**(*name=""*)
Create a *TarInfo* object.

classmethod `TarInfo.`**frombuf**(*buf*, *encoding*, *errors*)
Create and return a *TarInfo* object from string buffer *buf*.

Raises *HeaderError* if the buffer is invalid.

classmethod `TarInfo.`**fromtarfile**(*tarfile*)
Read the next member from the *TarFile* object *tarfile* and return it as a *TarInfo* object.

`TarInfo.`**tobuf**(*format=DEFAULT_FORMAT*, *encoding=ENCODING*, *errors='surrogateescape'*)
Create a string buffer from a *TarInfo* object. For information on the arguments see the constructor of the *TarFile* class.

Alterado na versão 3.2: Use `'surrogateescape'` as the default for the *errors* argument.

A `TarInfo` object has the following public data attributes:

`TarInfo.`**name**
Name of the archive member.

`TarInfo.`**size**
Size in bytes.

`TarInfo.`**mtime**
Time of last modification.

`TarInfo.`**mode**
Permission bits.

`TarInfo.`**type**
File type. *type* is usually one of these constants: `REGTYPE`, `AREGTYPE`, `LNKTYPE`, `SYMTYPE`, `DIRTYPE`, `FIFOTYPE`, `CONTTYPE`, `CHRTYPE`, `BLKTYPE`, `GNUTYPE_SPARSE`. To determine the type of a *TarInfo* object more conveniently, use the `is*()` methods below.

`TarInfo.`**linkname**
Name of the target file name, which is only present in *TarInfo* objects of type `LNKTYPE` and `SYMTYPE`.

`TarInfo.`**uid**
User ID of the user who originally stored this member.

`TarInfo.`**gid**
Group ID of the user who originally stored this member.

`TarInfo.`**uname**
User name.

`TarInfo.`**gname**
Group name.

`TarInfo.`**pax_headers**
A dictionary containing key-value pairs of an associated pax extended header.

A *TarInfo* object also provides some convenient query methods:

TarInfo.**isfile**()
Return *True* if the Tarinfo object is a regular file.

TarInfo.**isreg**()
Same as *isfile()*.

TarInfo.**isdir**()
Return *True* if it is a directory.

TarInfo.**issym**()
Return *True* if it is a symbolic link.

TarInfo.**islnk**()
Return *True* if it is a hard link.

TarInfo.**ischr**()
Return *True* if it is a character device.

TarInfo.**isblk**()
Return *True* if it is a block device.

TarInfo.**isfifo**()
Return *True* if it is a FIFO.

TarInfo.**isdev**()
Return *True* if it is one of character device, block device or FIFO.

13.6.3 Interface de Linha de Comando

Novo na versão 3.4.

The *tarfile* module provides a simple command-line interface to interact with tar archives.

If you want to create a new tar archive, specify its name after the *-c* option and then list the filename(s) that should be included:

```
$ python -m tarfile -c monty.tar  spam.txt eggs.txt
```

Passar um diretório também é aceitável:

```
$ python -m tarfile -c monty.tar life-of-brian_1979/
```

If you want to extract a tar archive into the current directory, use the *-e* option:

```
$ python -m tarfile -e monty.tar
```

You can also extract a tar archive into a different directory by passing the directory's name:

```
$ python -m tarfile -e monty.tar  other-dir/
```

For a list of the files in a tar archive, use the *-l* option:

```
$ python -m tarfile -l monty.tar
```

Opções de linha de comando

-l <tarfile>
--list <tarfile>
List files in a tarfile.

-c <tarfile> <source1> ... <sourceN>
--create <tarfile> <source1> ... <sourceN>
Create tarfile from source files.

-e <tarfile> [<output_dir>]
--extract <tarfile> [<output_dir>]
Extract tarfile into the current directory if *output_dir* is not specified.

-t <tarfile>
--test <tarfile>
Test whether the tarfile is valid or not.

-v, --verbose
Verbose output.

13.6.4 Exemplos

How to extract an entire tar archive to the current working directory:

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

How to extract a subset of a tar archive with *TarFile.extractall()* using a generator function instead of a list:

```
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

How to create an uncompressed tar archive from a list of filenames:

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

The same example using the `with` statement:

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
```

(continua na próxima página)

(continuação da página anterior)

```
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

How to read a gzip compressed tar archive and display some member information:

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print(tarinfo.name, "is", tarinfo.size, "bytes in size and is ", end="")
    if tarinfo.isreg():
        print("a regular file.")
    elif tarinfo.isdir():
        print("a directory.")
    else:
        print("something else.")
tar.close()
```

How to create an archive and reset the user information using the *filter* parameter in *TarFile.add()*:

```
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

13.6.5 Supported tar formats

There are three tar formats that can be created with the *tarfile* module:

- The POSIX.1-1988 ustar format (*USTAR_FORMAT*). It supports filenames up to a length of at best 256 characters and linknames up to 100 characters. The maximum file size is 8 GiB. This is an old and limited but widely supported format.
- The GNU tar format (*GNU_FORMAT*). It supports long filenames and linknames, files bigger than 8 GiB and sparse files. It is the de facto standard on GNU/Linux systems. *tarfile* fully supports the GNU tar extensions for long names, sparse file support is read-only.
- The POSIX.1-2001 pax format (*PAX_FORMAT*). It is the most flexible format with virtually no limits. It supports long filenames and linknames, large files and stores pathnames in a portable way. However, not all tar implementations today are able to handle pax archives properly.

  The *pax* format is an extension to the existing *ustar* format. It uses extra headers for information that cannot be stored otherwise. There are two flavours of pax headers: Extended headers only affect the subsequent file header, global headers are valid for the complete archive and affect all following files. All the data in a pax header is encoded in *UTF-8* for portability reasons.

There are some more variants of the tar format which can be read, but not created:

- The ancient V7 format. This is the first tar format from Unix Seventh Edition, storing only regular files and directories. Names must not be longer than 100 characters, there is no user/group name information. Some archives have miscalculated header checksums in case of fields with non-ASCII characters.
- The SunOS tar extended format. This format is a variant of the POSIX.1-2001 pax format, but is not compatible.

13.6.6 Unicode issues

The tar format was originally conceived to make backups on tape drives with the main focus on preserving file system information. Nowadays tar archives are commonly used for file distribution and exchanging archives over networks. One problem of the original format (which is the basis of all other formats) is that there is no concept of supporting different character encodings. For example, an ordinary tar archive created on a *UTF-8* system cannot be read correctly on a *Latin-1* system if it contains non-*ASCII* characters. Textual metadata (like filenames, linknames, user/group names) will appear damaged. Unfortunately, there is no way to autodetect the encoding of an archive. The pax format was designed to solve this problem. It stores non-ASCII metadata using the universal character encoding *UTF-8*.

The details of character conversion in `tarfile` are controlled by the *encoding* and *errors* keyword arguments of the `TarFile` class.

encoding defines the character encoding to use for the metadata in the archive. The default value is `sys.getfilesystemencoding()` or `'ascii'` as a fallback. Depending on whether the archive is read or written, the metadata must be either decoded or encoded. If *encoding* is not set appropriately, this conversion may fail.

The *errors* argument defines how characters are treated that cannot be converted. Possible values are listed in section *Error Handlers*. The default scheme is `'surrogateescape'` which Python also uses for its file system calls, see *Nomes de arquivos, argumentos de linha de comando e variáveis de ambiente*.

In case of `PAX_FORMAT` archives, *encoding* is generally not needed because all the metadata is stored using *UTF-8*. *encoding* is only used in the rare cases when binary pax headers are decoded or when strings with surrogate characters are stored.

CAPÍTULO 14

Formatos de Arquivo

Os módulos descritos neste capítulo analisam vários formatos de arquivo diversos que não são linguagens de marcação e não estão relacionados ao e-mail.

14.1 `csv` — Leiture e Escrita de arquivos CSV

Código Fonte: Lib/csv.py

O chamado formato CSV (Comma Separated Values) é o formato mais comum de importação e exportação de planilhas e bancos de dados. O formato CSV foi usado por muitos anos antes das tentativas de descrever o formato de maneira padronizada em **RFC 4180**. A falta de um padrão bem definido significa que existem diferenças sutis nos dados produzidos e consumidos por diferentes aplicativos. Essas diferenças podem tornar irritante o processamento de arquivos CSV de várias fontes. Ainda assim, enquanto os delimitadores e os caracteres de citação variam, o formato geral é semelhante o suficiente para que seja possível escrever um único módulo que possa manipular eficientemente esses dados, ocultando os detalhes da leitura e gravação dos dados do programador.

O módulo `csv` implementa classes para ler e gravar dados tabulares no formato CSV. Ele permite que os programadores digam "escreva esses dados no formato preferido pelo Excel" ou "leia os dados desse arquivo gerado pelo Excel", sem conhecer os detalhes precisos do formato CSV usado pelo Excel. Os programadores também podem descrever os formatos CSV entendidos por outros aplicativos ou definir seus próprios formatos CSV para fins especiais.

Os objetos de `reader` e `writer` do módulo `csv` leem e escrevem sequências. Os programadores também podem ler e gravar dados no formato de dicionário usando as classes `DictReader` e `DictWriter`.

Ver também:

PEP 305 - API de arquivo CSV A proposta de melhoria do Python que propôs essa adição ao Python.

14.1.1 Conteúdo do Módulo

O módulo `csv` define as seguintes funções:

`csv.`**`reader`**(*csvfile*, *dialect='excel'*, ***fmtparams*)

Retorna um objeto leitor que irá iterar sobre as linhas no *csvfile* fornecido. *csvfile* pode ser qualquer objeto que possua suporte ao protocolo *iterador* e retorne uma string sempre que o método `__next__()` for chamado — *arquivos objeto* e objetos lista são ambos adequado. Se *csvfile* for um objeto de arquivo, ele deverá ser aberto com `newline=''`.[1] Pode ser fornecido um parâmetro opcional *dialect*, usado para definir um conjunto de parâmetros específicos para um dialeto CSV específico. Pode ser uma instância de uma subclasse da classe `Dialect` ou uma das strings retornadas pela função `list_dialects()`. Os outros argumentos nomeados opcionais *fmtparams* podem ser dados para substituir parâmetros de formatação individuais no dialeto atual. Para detalhes completos sobre os parâmetros de dialeto e formatação, consulte a seção *Dialetos e parâmetros de formatação*.

Cada linha lida no arquivo csv é retornada como uma lista de cadeias. Nenhuma conversão automática de tipo de dados é executada, a menos que a opção de formato `QUOTE_NONNUMERIC` seja especificada (nesse caso, os campos não citados são transformados em flutuadores).

Um pequeno exemplo de utilização

```
>>> import csv
>>> with open('eggs.csv', newline='') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print(', '.join(row))
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

`csv.`**`writer`**(*csvfile*, *dialect='excel'*, ***fmtparams*)

Retorna um objeto de escrita responsável por converter os dados de usuário em strings delimitadas no objeto semelhante a um arquivo. *csvfile* pode ser qualquer objeto com um método `write()`. Se *csvfile* for um objeto de arquivo, ele deverá ser aberto com `newline=''`[1]. Pode ser fornecido um parâmetro opcional *dialect*, usado para definir um conjunto de parâmetros específicos para um dialeto CSV específico. Pode ser uma instância de uma subclasse da classe `Dialect` ou uma das strings retornadas pela função `list_dialects()`. Os outros argumentos nomeados opcionais *fmtparams* podem ser dados para substituir parâmetros de formatação individuais no dialeto atual. Para detalhes completos sobre os parâmetros de dialeto e formatação, consulte a seção *Dialetos e parâmetros de formatação*. Para tornar o mais fácil possível a interface com os módulos que implementam a API do DB, o valor `None` é escrito como uma string vazia. Embora isso não seja uma transformação reversível, torna mais fácil despejar valores de dados SQL NULL em arquivos CSV sem preprocessar os dados retornados de uma chamada `cursor.fetch*`. Todos os outros dados que não são de strings são codificados com `str()` antes de serem escritos.

Um pequeno exemplo de utilização

```
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

`csv.`**`register_dialect`**(*name*[, *dialect*[, ***fmtparams*]])

Associa *dialect* a *name*. *name* deve ser uma string. O dialeto pode ser especificado passando uma subclasse de

[1] Se `newline=''` não for especificado, as novas linhas incorporadas nos campos entre aspas não serão interpretadas corretamente, e nas plataformas que usam fim de linha `\r\n` na escrita, um `\r` extra será adicionado. Sempre deve ser seguro especificar `newline=''`, já que o módulo csv faz seu próprio tratamento de nova linha (*universal*).

Dialect ou por *fmtparams* argumentos nomeados, ou ambos, com argumentos nomeados substituindo os parâmetros do dialeto. Para detalhes completos sobre os parâmetros de dialeto e formatação, consulte a seção *Dialetos e parâmetros de formatação*.

csv.**unregister_dialect**(*name*)
Exclui o dialeto associado ao *name* do registro do dialeto. Um *Error* é levantado se *name* não for um nome de dialeto registrado.

csv.**get_dialect**(*name*)
Retorna o dialeto associado a *name*. Um *Error* é levantado se *name* não for um nome de dialeto registrado. Esta função retorna uma classe *Dialect* imutável.

csv.**list_dialects**()
Retorna os nomes de todos os dialetos registrados

csv.**field_size_limit**([*new_limit*])
Retorna o tamanho máximo atual do campo permitido pelo analisador sintático. Se *new_limit* for fornecido, este se tornará o novo limite.

O módulo *csv* define as seguintes classes:

class csv.**DictReader**(*f*, *fieldnames=None*, *restkey=None*, *restval=None*, *dialect='excel'*, **args*, ***kwds*)
Create an object that operates like a regular reader but maps the information in each row to an *OrderedDict* whose keys are given by the optional *fieldnames* parameter.

The *fieldnames* parameter is a *sequence*. If *fieldnames* is omitted, the values in the first row of file *f* will be used as the fieldnames. Regardless of how the fieldnames are determined, the ordered dictionary preserves their original ordering.

Se uma linha tiver mais campos que nomes de campo, os dados restantes serão colocados em uma lista e armazenados com o nome do campo especificado por *restkey* (o padrão é `None`). Se uma linha que não estiver em branco tiver menos campos que nomes de campo, os valores ausentes serão preenchidos com o valor *restval* (o padrão é `None`).

Todos os outros argumentos nomeados ou opcionais são passados para a instância subjacente de *reader*.

Alterado na versão 3.6: Linhas retornadas agora são do tipo `OrderedDict`.

Um pequeno exemplo de utilização

```
>>> import csv
>>> with open('names.csv', newline='') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Eric Idle
John Cleese

>>> print(row)
OrderedDict([('first_name', 'John'), ('last_name', 'Cleese')])
```

class csv.**DictWriter**(*f*, *fieldnames*, *restval=''*, *extrasaction='raise'*, *dialect='excel'*, **args*, ***kwds*)
Cria um objeto que funcione como um de escrita comum, mas mapeia dicionários nas linhas de saída. O parâmetro *fieldnames* é uma *sequência* de chaves que identificam a ordem na qual os valores no dicionário transmitidos para o método `writerow()` são escritos no arquivo *f*. O parâmetro opcional *restval* especifica o valor a ser escrito se o dicionário estiver com falta de uma chave em *fieldnames*. Se o dicionário transmitido para o método `writerow()` contiver uma chave não encontrada em *fieldnames*, o parâmetro opcional *extrasaction* indica qual ação executar. Se estiver definido como `'raise'`, o valor padrão, a *ValueError* é levantada. Se estiver definido como `'ignore'`, valores extras no dicionário serão ignorados. Quaisquer outros argumentos nomeados ou opcionais são passados para a instância subjacente de *writer*.

Observe que, diferentemente da classe *DictReader*, o parâmetro *fieldnames* da classe *DictWriter* não é opcional.

Um pequeno exemplo de utilização

```
import csv

with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

class csv.**Dialect**
A classe *Dialect* é uma classe de contêiner baseada principalmente em seus atributos, que são usados para definir os parâmetros para uma instância específica *reader* ou *writer*.

class csv.**excel**
A classe *excel* define as propriedades usuais de um arquivo CSV gerado pelo Excel. Ele é registrado com o nome do dialeto `'excel'`.

class csv.**excel_tab**
A classe *excel_tab* define as propriedades usuais de um arquivo delimitado por TAB gerado pelo Excel. Ela é registrado com o nome do dialeto `'excel-tab'`.

class csv.**unix_dialect**
A classe *unix_dialect* define as propriedades usuais de um arquivo CSV gerado em sistemas UNIX, ou seja, usando `'\n'` como terminador de linha e citando todos os campos. É registrado com o nome do dialeto `'unix'`.

Novo na versão 3.2.

class csv.**Sniffer**
A classe *Sniffer* é usada para deduzir o formato de um arquivo CSV.

A classe *Sniffer* fornece dois métodos:

sniff(*sample*, *delimiters=None*)
Analisa a *sample* fornecida e retorna uma subclasse *Dialect*, refletindo os parâmetros encontrados. Se o parâmetro opcional *delimiters* for fornecido, ele será interpretado como uma string contendo possíveis caracteres válidos de delimitador.

has_header(*sample*)
Analisa o texto de amostra (presumivelmente no formato CSV) e retorna *True* se a primeira linha parecer ser uma série de cabeçalhos de coluna.

Um exemplo para uso de *Sniffer*:

```
with open('example.csv', newline='') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

O módulo *csv* define as seguintes constantes:

csv.**QUOTE_ALL**
Instrui objetos *writer* a colocar aspas em todos os campos.

csv.**QUOTE_MINIMAL**
Instrui objetos *writer* a colocar aspas apenas nos campos que contêm caracteres especiais como *delimiters*, *quotechar* ou qualquer um dos caracteres em *lineterminator*.

csv.**QUOTE_NONNUMERIC**
Instrui objetos *writer* a colocar aspas em todos os campos não numéricos.

Instrui o leitor a converter todos os campos não citados no tipo *float*.

csv.**QUOTE_NONE**
Instrui objetos *writer* a nunca colocar aspas nos campos. Quando o *delimiter* atual ocorre nos dados de saída, é precedido pelo caractere *escapechar* atual. Se *escapeechar* não estiver definido, o escritor levantará *Error* se algum caractere que exija escape for encontrado.

Instrui *reader* a não executar nenhum processamento especial de caracteres de aspas.

O módulo *csv* define a seguinte exceção:

exception csv.**Error**
Levantada por qualquer uma das funções quando um erro é detectado.

14.1.2 Dialetos e parâmetros de formatação

Para facilitar a especificação do formato dos registros de entrada e saída, parâmetros específicos de formatação são agrupados em dialetos. Um dialeto é uma subclasse da classe *Dialect* com um conjunto de métodos específicos e um único método `validate()`. Ao criar objetos *reader* ou *writer*, o programador pode especificar uma string ou uma subclasse da classe *Dialect* como parâmetro de dialeto. Além do parâmetro *dialect*, ou em vez do parâmetro *dialect*, o programador também pode especificar parâmetros de formatação individuais, com os mesmos nomes dos atributos definidos abaixo para a classe *Dialect*.

Os dialetos possuem suporte aos seguintes atributos:

Dialect.**delimiter**
Uma string de um caractere usada para separar campos. O padrão é `','`.

Dialect.**doublequote**
Controla como as instâncias de *quotechar* que aparecem dentro de um campo devem estar entre aspas. Quando *True*, o caractere é dobrado. Quando *False*, o *escapechar* é usado como um prefixo para o *quotechar*. O padrão é *True*.

Na saída, se *doublequote* for *False* e não *escapeechar* for definido, *Error* é levantada se um *quotechar* é encontrado em um campo.

Dialect.**escapechar**
Uma string usada pelo escritor para escapar do *delimitator* se *quoting* estiver definida como *QUOTE_NONE* e o *quotechar* se *quoting* for *False*. Na leitura, o *escapechar* remove qualquer significado especial do caractere seguinte. O padrão é *None*, que desativa o escape.

Dialect.**lineterminator**
A string usada para terminar as linhas produzidas pelo *writer*. O padrão é `'\r\n'`.

Nota: A *reader* é codificada para reconhecer `'\r'` ou `'\n'` como fim de linha e ignora *lineterminator*. Esse comportamento pode mudar no futuro.

Dialect.**quotechar**
Uma string de um caractere usada para citar campos contendo caracteres especiais, como *delimiter* ou *quotechar*, ou que contêm caracteres de nova linha. O padrão é `'"'`.

Dialect.**quoting**
: Controla quando as aspas devem ser geradas pelo escritor e reconhecidas pelo leitor. Ele pode assumir qualquer uma das constantes QUOTE_* (consulte a seção *Conteúdo do Módulo*) e o padrão é *QUOTE_MINIMAL*.

Dialect.**skipinitialspace**
: Quando *True*, os espaços em branco imediatamente após o *delimiter* são ignorados. O padrão é *False*.

Dialect.**strict**
: Quando True, levanta a exceção *Error* em uma entrada CSV ruim. O padrão é False.

14.1.3 Objetos Reader

Os objetos Reader (instâncias *DictReader* e objetos retornados pela função *reader()*) têm os seguintes métodos públicos:

csvreader.**__next__**()
: Retorna a próxima linha do objeto iterável do leitor como uma lista (se o objeto foi retornado de *reader()*) ou um dict (se for uma instância de *DictReader*), analisada de acordo com o dialeto atual. Normalmente você deve chamar isso como next(reader).

Os objetos Reader possuem os seguintes atributos públicos:

csvreader.**dialect**
: Uma descrição somente leitura do dialeto em uso pelo analisador sintático.

csvreader.**line_num**
: O número de linhas lidas no iterador de origem. Não é o mesmo que o número de registros retornados, pois os registros podem abranger várias linhas.

Os objetos DictReader têm o seguinte atributo público:

csvreader.**fieldnames**
: Se não for passado como parâmetro ao criar o objeto, esse atributo será inicializado no primeiro acesso ou quando o primeiro registro for lido no arquivo.

14.1.4 Objetos Writer

Objetos Writer (instâncias e objetos *DictWriter* retornados pela função *writer()*) possuem os seguintes métodos públicos. Uma *row* deve ser iterável de strings ou números para objetos Writer e um dicionário mapeando nomes de campos para strings ou números (passando-os por *str()* primeiro) para *DictWriter*. Observe que números complexos são escritos cercados por parênteses. Isso pode causar alguns problemas para outros programas que leem arquivos CSV (supondo que eles suportem números complexos).

csvwriter.**writerow**(*row*)
: Write the *row* parameter to the writer's file object, formatted according to the current dialect.

  Alterado na versão 3.5: Adicionado suporte a iteráveis arbitrários.

csvwriter.**writerows**(*rows*)
: Escreve todos os elementos em *rows* (um iterável de objetos *row*, conforme descrito acima) no objeto arquivo do escritor, formatado de acordo com o dialeto atual.

Os objetos Writer têm o seguinte atributo público:

csvwriter.**dialect**
: Uma descrição somente leitura do dialeto em uso pelo escritor.

Os objetos DictWriter têm o seguinte método público:

DictWriter.**writeheader**()
Write a row with the field names (as specified in the constructor).

Novo na versão 3.2.

14.1.5 Exemplos

O exemplo mais simples de leitura de um arquivo CSV:

```
import csv
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Lendo um arquivo com um formato alternativo:

```
import csv
with open('passwd', newline='') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print(row)
```

O exemplo de escrita possível mais simples possível é:

```
import csv
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

Como *open()* é usado para abrir um arquivo CSV para leitura, o arquivo será decodificado por padrão em unicode usando a codificação padrão do sistema (consulte *locale.getpreferredencoding()*). Para decodificar um arquivo usando uma codificação diferente, use o argumento `encoding` do open:

```
import csv
with open('some.csv', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

O mesmo se aplica à escrita em algo diferente da codificação padrão do sistema: especifique o argumento de codificação ao abrir o arquivo de saída.

Registrando um novo dialeto:

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', newline='') as f:
    reader = csv.reader(f, 'unixpwd')
```

Um uso um pouco mais avançado do leitor — capturando e relatando erros:

```
import csv, sys
filename = 'some.csv'
with open(filename, newline='') as f:
    reader = csv.reader(f)
    try:
```

(continua na próxima página)

(continuação da página anterior)

```
        for row in reader:
            print(row)
    except csv.Error as e:
        sys.exit('file {}, line {}: {}'.format(filename, reader.line_num, e))
```

E embora o módulo não tenha suporte diretamente à análise sintática de strings, isso pode ser feito facilmente:

```
import csv
for row in csv.reader(['one,two,three']):
    print(row)
```

14.2 `configparser` — Configuration file parser

Código Fonte: Lib/configparser.py

This module provides the *ConfigParser* class which implements a basic configuration language which provides a structure similar to what's found in Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

Nota: This library does *not* interpret or write the value-type prefixes used in the Windows Registry extended version of INI syntax.

Ver também:

Module *shlex* Support for creating Unix shell-like mini-languages which can be used as an alternate format for application configuration files.

Module *json* The json module implements a subset of JavaScript syntax which can also be used for this purpose.

14.2.1 Quick Start

Let's take a very basic configuration file that looks like this:

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[bitbucket.org]
User = hg

[topsecret.server.com]
Port = 50022
ForwardX11 = no
```

The structure of INI files is described *in the following section*. Essentially, the file consists of sections, each of which contains keys with values. *configparser* classes can read and write such files. Let's start by creating the above configuration file programmatically.

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45',
...                      'Compression': 'yes',
...                      'CompressionLevel': '9'}
>>> config['bitbucket.org'] = {}
>>> config['bitbucket.org']['User'] = 'hg'
>>> config['topsecret.server.com'] = {}
>>> topsecret = config['topsecret.server.com']
>>> topsecret['Port'] = '50022'     # mutates the parser
>>> topsecret['ForwardX11'] = 'no'  # same here
>>> config['DEFAULT']['ForwardX11'] = 'yes'
>>> with open('example.ini', 'w') as configfile:
...   config.write(configfile)
...
```

As you can see, we can treat a config parser much like a dictionary. There are differences, *outlined later*, but the behavior is very close to what you would expect from a dictionary.

Now that we have created and saved a configuration file, let's read it back and explore the data it holds.

```
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['bitbucket.org', 'topsecret.server.com']
>>> 'bitbucket.org' in config
True
>>> 'bytebong.com' in config
False
>>> config['bitbucket.org']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.com']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['bitbucket.org']:
...     print(key)
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['bitbucket.org']['ForwardX11']
'yes'
```

As we can see above, the API is pretty straightforward. The only bit of magic involves the `DEFAULT` section which provides default values for all other sections[1]. Note also that keys in sections are case-insensitive and stored in lowercase[1].

[1] Config parsers allow for heavy customization. If you are interested in changing the behaviour outlined by the footnote reference, consult the *Customizing Parser Behaviour* section.

14.2.2 Supported Datatypes

Config parsers do not guess datatypes of values in configuration files, always storing them internally as strings. This means that if you need other datatypes, you should convert on your own:

```
>>> int(topsecret['Port'])
50022
>>> float(topsecret['CompressionLevel'])
9.0
```

Since this task is so common, config parsers provide a range of handy getter methods to handle integers, floats and booleans. The last one is the most interesting because simply passing the value to `bool()` would do no good since `bool('False')` is still `True`. This is why config parsers also provide *getboolean()*. This method is case-insensitive and recognizes Boolean values from `'yes'`/`'no'`, `'on'`/`'off'`, `'true'`/`'false'` and `'1'`/`'0'`[1]. For example:

```
>>> topsecret.getboolean('ForwardX11')
False
>>> config['bitbucket.org'].getboolean('ForwardX11')
True
>>> config.getboolean('bitbucket.org', 'Compression')
True
```

Apart from *getboolean()*, config parsers also provide equivalent *getint()* and *getfloat()* methods. You can register your own converters and customize the provided ones.[1]

14.2.3 Fallback Values

As with a dictionary, you can use a section's `get()` method to provide fallback values:

```
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

Please note that default values have precedence over fallback values. For instance, in our example the `'CompressionLevel'` key was specified only in the `'DEFAULT'` section. If we try to get it from the section `'topsecret.server.com'`, we will always get the default, even if we specify a fallback:

```
>>> topsecret.get('CompressionLevel', '3')
'9'
```

One more thing to be aware of is that the parser-level `get()` method provides a custom, more complex interface, maintained for backwards compatibility. When using this method, a fallback value can be provided via the `fallback` keyword-only argument:

```
>>> config.get('bitbucket.org', 'monster',
...            fallback='No such things as monsters')
'No such things as monsters'
```

The same `fallback` argument can be used with the *getint()*, *getfloat()* and *getboolean()* methods, for example:

```
>>> 'BatchMode' in topsecret
False
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```

14.2.4 Estrutura dos arquivos INI

A configuration file consists of sections, each led by a `[section]` header, followed by key/value entries separated by a specific string (= or `:` by default[1]). By default, section names are case sensitive but keys are not[1]. Leading and trailing whitespace is removed from keys and values. Values can be omitted, in which case the key/value delimiter may also be left out. Values can also span multiple lines, as long as they are indented deeper than the first line of the value. Depending on the parser's mode, blank lines may be treated as parts of multiline values or ignored.

Configuration files may include comments, prefixed by specific characters (`#` and `;` by default[1]). Comments may appear on their own on an otherwise empty line, possibly indented.[1]

Por exemplo:

```
[Simple Values]
key=value
spaces in keys=allowed
spaces in values=allowed as well
spaces around the delimiter = obviously
you can also use : to delimit keys from values

[All Values Are Strings]
values like this: 1000000
or this: 3.14159265359
are they treated as numbers? : no
integers, floats and booleans are held as: strings
can use the API to get converted values directly: true

[Multiline Values]
chorus: I'm a lumberjack, and I'm okay
    I sleep all night and I work all day

[No Values]
key_without_value
empty string value here =

[You can use comments]
# like this
; or this

# By default only in an empty line.
# Inline comments can be harmful because they prevent users
# from using the delimiting characters as parts of values.
# That being said, this can be customized.

    [Sections Can Be Indented]
        can_values_be_as_well = True
        does_that_mean_anything_special = False
        purpose = formatting for readability
```

(continua na próxima página)

(continuação da página anterior)

```
        multiline_values = are
            handled just fine as
            long as they are indented
            deeper than the first line
            of a value
        # Did I mention we can indent comments, too?
```

14.2.5 Interpolation of values

On top of the core functionality, *ConfigParser* supports interpolation. This means values can be preprocessed before returning them from `get()` calls.

class configparser.**BasicInterpolation**

The default implementation used by *ConfigParser*. It enables values to contain format strings which refer to other values in the same section, or values in the special default section[1]. Additional default values can be provided on initialization.

Por exemplo:

```
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures

[Escape]
gain: 80%%  # use a %% to escape the % sign (% is the only character that needs
↪to be escaped)
```

In the example above, *ConfigParser* with *interpolation* set to `BasicInterpolation()` would resolve `%(home_dir)s` to the value of `home_dir` (`/Users` in this case). `%(my_dir)s` in effect would resolve to `/Users/lumberjack`. All interpolations are done on demand so keys used in the chain of references do not have to be specified in any specific order in the configuration file.

With `interpolation` set to `None`, the parser would simply return `%(my_dir)s/Pictures` as the value of `my_pictures` and `%(home_dir)s/lumberjack` as the value of `my_dir`.

class configparser.**ExtendedInterpolation**

An alternative handler for interpolation which implements a more advanced syntax, used for instance in `zc.buildout`. Extended interpolation is using `${section:option}` to denote a value from a foreign section. Interpolation can span multiple levels. For convenience, if the `section:` part is omitted, interpolation defaults to the current section (and possibly the default values from the special section).

For example, the configuration specified above with basic interpolation, would look like this with extended interpolation:

```
[Paths]
home_dir: /Users
my_dir: ${home_dir}/lumberjack
my_pictures: ${my_dir}/Pictures

[Escape]
cost: $$80  # use a $$ to escape the $ sign ($ is the only character that needs
↪to be escaped)
```

Values from other sections can be fetched as well:

```
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.2
path: ${Common:system_dir}/Library/Frameworks/

[Arthur]
nickname: Two Sheds
last_name: Jackson
my_dir: ${Common:home_dir}/twosheds
my_pictures: ${my_dir}/Pictures
python_dir: ${Frameworks:path}/Python/Versions/${Frameworks:Python}
```

14.2.6 Mapping Protocol Access

Novo na versão 3.2.

Mapping protocol access is a generic name for functionality that enables using custom objects as if they were dictionaries. In case of *configparser*, the mapping interface implementation is using the `parser['section']['option']` notation.

`parser['section']` in particular returns a proxy for the section's data in the parser. This means that the values are not copied but they are taken from the original parser on demand. What's even more important is that when values are changed on a section proxy, they are actually mutated in the original parser.

configparser objects behave as close to actual dictionaries as possible. The mapping interface is complete and adheres to the *MutableMapping* ABC. However, there are a few differences that should be taken into account:

- By default, all keys in sections are accessible in a case-insensitive manner[1]. E.g. `for option in parser["section"]` yields only `optionxform`'ed option key names. This means lowercased keys by default. At the same time, for a section that holds the key `'a'`, both expressions return `True`:

```
"a" in parser["section"]
"A" in parser["section"]
```

- All sections include `DEFAULTSECT` values as well which means that `.clear()` on a section may not leave the section visibly empty. This is because default values cannot be deleted from the section (because technically they are not there). If they are overridden in the section, deleting causes the default value to be visible again. Trying to delete a default value causes a *KeyError*.
- `DEFAULTSECT` cannot be removed from the parser:
  - trying to delete it raises *ValueError*,
  - `parser.clear()` leaves it intact,
  - `parser.popitem()` never returns it.
- `parser.get(section, option, **kwargs)` - the second argument is **not** a fallback value. Note however that the section-level `get()` methods are compatible both with the mapping protocol and the classic configparser API.
- `parser.items()` is compatible with the mapping protocol (returns a list of *section_name*, *section_proxy* pairs including the DEFAULTSECT). However, this method can also be invoked with arguments: `parser.`

`items(section, raw, vars)`. The latter call returns a list of *option*, *value* pairs for a specified `section`, with all interpolations expanded (unless `raw=True` is provided).

The mapping protocol is implemented on top of the existing legacy API so that subclasses overriding the original interface still should have mappings working as expected.

14.2.7 Customizing Parser Behaviour

There are nearly as many INI format variants as there are applications using it. *configparser* goes a long way to provide support for the largest sensible set of INI styles available. The default functionality is mainly dictated by historical background and it's very likely that you will want to customize some of the features.

The most common way to change the way a specific config parser works is to use the `__init__()` options:

- *defaults*, default value: `None`

  This option accepts a dictionary of key-value pairs which will be initially put in the `DEFAULT` section. This makes for an elegant way to support concise configuration files that don't specify values which are the same as the documented default.

  Hint: if you want to specify default values for a specific section, use `read_dict()` before you read the actual file.

- *dict_type*, default value: *collections.OrderedDict*

  This option has a major impact on how the mapping protocol will behave and how the written configuration files look. With the default ordered dictionary, every section is stored in the order they were added to the parser. Same goes for options within sections.

  An alternative dictionary type can be used for example to sort sections and options on write-back. You can also use a regular dictionary for performance reasons.

  Please note: there are ways to add a set of key-value pairs in a single operation. When you use a regular dictionary in those operations, the order of the keys will be ordered because dict preserves order from Python 3.7. For example:

```
>>> parser = configparser.ConfigParser()
>>> parser.read_dict({'section1': {'key1': 'value1',
...                                'key2': 'value2',
...                                'key3': 'value3'},
...                   'section2': {'keyA': 'valueA',
...                                'keyB': 'valueB',
...                                'keyC': 'valueC'},
...                   'section3': {'foo': 'x',
...                                'bar': 'y',
...                                'baz': 'z'}
... })
>>> parser.sections()
['section1', 'section2', 'section3']
>>> [option for option in parser['section3']]
['foo', 'bar', 'baz']
```

- *allow_no_value*, default value: `False`

  Some configuration files are known to include settings without values, but which otherwise conform to the syntax supported by *configparser*. The *allow_no_value* parameter to the constructor can be used to indicate that such values should be accepted:

```
>>> import configparser
```

(continua na próxima página)

(continuação da página anterior)

```
>>> sample_config = """
... [mysqld]
...   user = mysql
...   pid-file = /var/run/mysqld/mysqld.pid
...   skip-external-locking
...   old_passwords = 1
...   skip-bdb
...   # we don't need ACID today
...   skip-innodb
... """
>>> config = configparser.ConfigParser(allow_no_value=True)
>>> config.read_string(sample_config)

>>> # Settings with values are treated as before:
>>> config["mysqld"]["user"]
'mysql'

>>> # Settings without values provide None:
>>> config["mysqld"]["skip-bdb"]

>>> # Settings which aren't specified still raise an error:
>>> config["mysqld"]["does-not-exist"]
Traceback (most recent call last):
  ...
KeyError: 'does-not-exist'
```

- *delimiters*, default value: `('=', ':')`

  Delimiters are substrings that delimit keys from values within a section. The first occurrence of a delimiting substring on a line is considered a delimiter. This means values (but not keys) can contain the delimiters.

  See also the *space_around_delimiters* argument to *ConfigParser.write()*.

- *comment_prefixes*, default value: `('#', ';')`

- *inline_comment_prefixes*, default value: `None`

  Comment prefixes are strings that indicate the start of a valid comment within a config file. *comment_prefixes* are used only on otherwise empty lines (optionally indented) whereas *inline_comment_prefixes* can be used after every valid value (e.g. section names, options and empty lines as well). By default inline comments are disabled and `'#'` and `';'` are used as prefixes for whole line comments.

  Alterado na versão 3.2: In previous versions of *configparser* behaviour matched `comment_prefixes=('#',';')` and `inline_comment_prefixes=(';',)`.

  Please note that config parsers don't support escaping of comment prefixes so using *inline_comment_prefixes* may prevent users from specifying option values with characters used as comment prefixes. When in doubt, avoid setting *inline_comment_prefixes*. In any circumstances, the only way of storing comment prefix characters at the beginning of a line in multiline values is to interpolate the prefix, for example:

```
>>> from configparser import ConfigParser, ExtendedInterpolation
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> # the default BasicInterpolation could be used as well
>>> parser.read_string("""
... [DEFAULT]
... hash = #
...
... [hashes]
```

(continua na próxima página)

(continuação da página anterior)

```
... shebang =
...   ${hash}!/usr/bin/env python
...   ${hash} -*- coding: utf-8 -*-
...
... extensions =
...   enabled_extension
...   another_extension
...   #disabled_by_comment
...   yet_another_extension
...
... interpolation not necessary = if # is not at line start
... even in multiline values = line #1
...   line #2
...   line #3
... """)
>>> print(parser['hashes']['shebang'])

#!/usr/bin/env python
# -*- coding: utf-8 -*-
>>> print(parser['hashes']['extensions'])

enabled_extension
another_extension
yet_another_extension
>>> print(parser['hashes']['interpolation not necessary'])
if # is not at line start
>>> print(parser['hashes']['even in multiline values'])
line #1
line #2
line #3
```

- *strict*, o valor padrão é: `True`

  When set to `True`, the parser will not allow for any section or option duplicates while reading from a single source (using `read_file()`, `read_string()` or `read_dict()`). It is recommended to use strict parsers in new applications.

  Alterado na versão 3.2: In previous versions of *configparser* behaviour matched `strict=False`.

- *empty_lines_in_values*, valor padrão: `True`

  In config parsers, values can span multiple lines as long as they are indented more than the key that holds them. By default parsers also let empty lines to be parts of values. At the same time, keys can be arbitrarily indented themselves to improve readability. In consequence, when configuration files get big and complex, it is easy for the user to lose track of the file structure. Take for instance:

```
[Section]
key = multiline
  value with a gotcha

 this = is still a part of the multiline value of 'key'
```

  This can be especially problematic for the user to see if she's using a proportional font to edit the file. That is why when your application does not need values with empty lines, you should consider disallowing them. This will make empty lines split keys every time. In the example above, it would produce two keys, `key` and `this`.

- *default_section*, valor default: `configparser.DEFAULTSECT` (isto é: `"DEFAULT"`)

  The convention of allowing a special section of default values for other sections or interpolation purposes is a

powerful concept of this library, letting users create complex declarative configurations. This section is normally called `"DEFAULT"` but this can be customized to point to any other valid section name. Some typical values include: `"general"` or `"common"`. The name provided is used for recognizing default sections when reading from any source and is used when writing configuration back to a file. Its current value can be retrieved using the `parser_instance.default_section` attribute and may be modified at runtime (i.e. to convert files from one format to another).

- *interpolation*, default value: `configparser.BasicInterpolation`

  Interpolation behaviour may be customized by providing a custom handler through the *interpolation* argument. `None` can be used to turn off interpolation completely, `ExtendedInterpolation()` provides a more advanced variant inspired by `zc.buildout`. More on the subject in the *dedicated documentation section*. *`RawConfigParser`* has a default value of `None`.

- *converters*, valor default : not set

  Config parsers provide option value getters that perform type conversion. By default *`getint()`*, *`getfloat()`*, and *`getboolean()`* are implemented. Should other getters be desirable, users may define them in a subclass or pass a dictionary where each key is a name of the converter and each value is a callable implementing said conversion. For instance, passing `{'decimal': decimal.Decimal}` would add `getdecimal()` on both the parser object and all section proxies. In other words, it will be possible to write both `parser_instance.getdecimal('section', 'key', fallback=0)` and `parser_instance['section'].getdecimal('key', 0)`.

  If the converter needs to access the state of the parser, it can be implemented as a method on a config parser subclass. If the name of this method starts with `get`, it will be available on all section proxies, in the dict-compatible form (see the `getdecimal()` example above).

More advanced customization may be achieved by overriding default values of these parser attributes. The defaults are defined on the classes, so they may be overridden by subclasses or by attribute assignment.

`ConfigParser.`**`BOOLEAN_STATES`**

By default when using *`getboolean()`*, config parsers consider the following values `True`: `'1'`, `'yes'`, `'true'`, `'on'` and the following values `False`: `'0'`, `'no'`, `'false'`, `'off'`. You can override this by specifying a custom dictionary of strings and their Boolean outcomes. For example:

```
>>> custom = configparser.ConfigParser()
>>> custom['section1'] = {'funky': 'nope'}
>>> custom['section1'].getboolean('funky')
Traceback (most recent call last):
...
ValueError: Not a boolean: nope
>>> custom.BOOLEAN_STATES = {'sure': True, 'nope': False}
>>> custom['section1'].getboolean('funky')
False
```

Other typical Boolean pairs include `accept`/`reject` or `enabled`/`disabled`.

`ConfigParser.`**`optionxform`**`(`*`option`*`)`

This method transforms option names on every read, get, or set operation. The default converts the name to lowercase. This also means that when a configuration file gets written, all keys will be lowercase. Override this method if that's unsuitable. For example:

```
>>> config = """
... [Section1]
... Key = Value
...
... [Section2]
... AnotherKey = Value
```

(continua na próxima página)

(continuação da página anterior)

```
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> list(typical['Section1'].keys())
['key']
>>> list(typical['Section2'].keys())
['anotherkey']
>>> custom = configparser.RawConfigParser()
>>> custom.optionxform = lambda option: option
>>> custom.read_string(config)
>>> list(custom['Section1'].keys())
['Key']
>>> list(custom['Section2'].keys())
['AnotherKey']
```

Nota: The optionxform function transforms option names to a canonical form. This should be an idempotent function: if the name is already in canonical form, it should be returned unchanged.

`ConfigParser.`**`SECTCRE`**

A compiled regular expression used to parse section headers. The default matches `[section]` to the name `"section"`. Whitespace is considered part of the section name, thus `[ larch ]` will be read as a section of name `" larch "`. Override this attribute if that's unsuitable. For example:

```
>>> import re
>>> config = """
... [Section 1]
... option = value
...
... [  Section 2  ]
... another = val
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> typical.sections()
['Section 1', '  Section 2  ']
>>> custom = configparser.ConfigParser()
>>> custom.SECTCRE = re.compile(r"\[ *(?P<header>[^]]+?) *\]")
>>> custom.read_string(config)
>>> custom.sections()
['Section 1', 'Section 2']
```

Nota: While ConfigParser objects also use an `OPTCRE` attribute for recognizing option lines, it's not recommended to override it because that would interfere with constructor options *allow_no_value* and *delimiters*.

14.2.8 Exemplos de APIs legadas

Mainly because of backwards compatibility concerns, *configparser* provides also a legacy API with explicit `get`/`set` methods. While there are valid use cases for the methods outlined below, mapping protocol access is preferred for new projects. The legacy API is at times more advanced, low-level and downright counterintuitive.

An example of writing to a configuration file:

```
import configparser

config = configparser.RawConfigParser()

# Please note that using RawConfigParser's set functions, you can assign
# non-string values to keys internally, but will receive an error when
# attempting to write to a file or when you get it in non-raw mode. Setting
# values using the mapping protocol or ConfigParser's set() does not allow
# such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'w') as configfile:
    config.write(configfile)
```

An example of reading the configuration file again:

```
import configparser

config = configparser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective types
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print(a_float + an_int)

# Notice that the next output does not interpolate '%(bar)s' or '%(baz)s'.
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print(config.get('Section1', 'foo'))
```

To get interpolation, use *ConfigParser*:

```
import configparser

cfg = configparser.ConfigParser()
cfg.read('example.cfg')

# Set the optional *raw* argument of get() to True if you wish to disable
# interpolation in a single get operation.
print(cfg.get('Section1', 'foo', raw=False))  # -> "Python is fun!"
print(cfg.get('Section1', 'foo', raw=True))   # -> "%(bar)s is %(baz)s!"
```

(continua na próxima página)

(continuação da página anterior)

```
# The optional *vars* argument is a dict with members that will take
# precedence in interpolation.
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation',
                                       'baz': 'evil'}))

# The optional *fallback* argument can be used to provide a fallback value
print(cfg.get('Section1', 'foo'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'foo', fallback='Monty is not.'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'monster', fallback='No such things as monsters.'))
      # -> "No such things as monsters."

# A bare print(cfg.get('Section1', 'monster')) would raise NoOptionError
# but we can also use:

print(cfg.get('Section1', 'monster', fallback=None))
      # -> None
```

Default values are available in both types of ConfigParsers. They are used in interpolation if an option used is not defined elsewhere.

```
import configparser

# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each
config = configparser.ConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print(config.get('Section1', 'foo'))      # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print(config.get('Section1', 'foo'))      # -> "Life is hard!"
```

14.2.9 ConfigParser Objects

class configparser.**ConfigParser**(*defaults=None*, *dict_type=collections.OrderedDict*, *allow_no_value=False*, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*, *interpolation=BasicInterpolation()*, *converters={}*)

The main configuration parser. When *defaults* is given, it is initialized into the dictionary of intrinsic defaults. When *dict_type* is given, it will be used to create the dictionary objects for the list of sections, for the options within a section, and for the default values.

When *delimiters* is given, it is used as the set of substrings that divide keys from values. When *comment_prefixes* is given, it will be used as the set of substrings that prefix comments in otherwise empty lines. Comments can be indented. When *inline_comment_prefixes* is given, it will be used as the set of substrings that prefix comments in non-empty lines.

When *strict* is `True` (the default), the parser won't allow for any section or option duplicates while reading from a single source (file, string or dictionary), raising `DuplicateSectionError` or `DuplicateOptionError`.

When *empty_lines_in_values* is `False` (default: `True`), each empty line marks the end of an option. Otherwise, internal empty lines of a multiline option are kept as part of the value. When *allow_no_value* is `True` (default: `False`), options without values are accepted; the value held for these is `None` and they are serialized without the trailing delimiter.

When *default_section* is given, it specifies the name for the special section holding default values for other sections and interpolation purposes (normally named `"DEFAULT"`). This value can be retrieved and changed on runtime using the `default_section` instance attribute.

Interpolation behaviour may be customized by providing a custom handler through the *interpolation* argument. `None` can be used to turn off interpolation completely, `ExtendedInterpolation()` provides a more advanced variant inspired by `zc.buildout`. More on the subject in the *dedicated documentation section*.

All option names used in interpolation will be passed through the *optionxform()* method just like any other option name reference. For example, using the default implementation of *optionxform()* (which converts option names to lower case), the values `foo %(bar)s` and `foo %(BAR)s` are equivalent.

When *converters* is given, it should be a dictionary where each key represents the name of a type converter and each value is a callable implementing the conversion from string to the desired datatype. Every converter gets its own corresponding `get*()` method on the parser object and section proxies.

Alterado na versão 3.1: The default *dict_type* is *collections.OrderedDict*.

Alterado na versão 3.2: *allow_no_value*, *delimiters*, *comment_prefixes*, *strict*, *empty_lines_in_values*, *default_section* and *interpolation* were added.

Alterado na versão 3.5: The *converters* argument was added.

Alterado na versão 3.7: The *defaults* argument is read with *read_dict()*, providing consistent behavior across the parser: non-string keys and values are implicitly converted to strings.

defaults()
Return a dictionary containing the instance-wide defaults.

sections()
Return a list of the sections available; the *default section* is not included in the list.

add_section(*section*)
Add a section named *section* to the instance. If a section by the given name already exists, *DuplicateSectionError* is raised. If the *default section* name is passed, *ValueError* is raised. The name of the section must be a string; if not, *TypeError* is raised.

Alterado na versão 3.2: Non-string section names raise *TypeError*.

has_section(*section*)
Indicates whether the named *section* is present in the configuration. The *default section* is not acknowledged.

options(*section*)
Return a list of options available in the specified *section*.

has_option(*section*, *option*)
If the given *section* exists, and contains the given *option*, return *True*; otherwise return *False*. If the specified *section* is *None* or an empty string, DEFAULT is assumed.

read(*filenames*, *encoding=None*)
Attempt to read and parse an iterable of filenames, returning a list of filenames which were successfully parsed.

If *filenames* is a string, a *bytes* object or a *path-like object*, it is treated as a single filename. If a file named in *filenames* cannot be opened, that file will be ignored. This is designed so that you can specify an iterable of potential configuration file locations (for example, the current directory, the user's home directory, and some system-wide directory), and all existing configuration files in the iterable will be read.

If none of the named files exist, the *ConfigParser* instance will contain an empty dataset. An application which requires initial values to be loaded from a file should load the required file or files using *read_file()* before calling *read()* for any optional files:

```
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')],
            encoding='cp1250')
```

Novo na versão 3.2: The *encoding* parameter. Previously, all files were read using the default encoding for *open()*.

Novo na versão 3.6.1: The *filenames* parameter accepts a *path-like object*.

Novo na versão 3.7: The *filenames* parameter accepts a *bytes* object.

read_file(*f*, *source=None*)
Read and parse configuration data from *f* which must be an iterable yielding Unicode strings (for example files opened in text mode).

Optional argument *source* specifies the name of the file being read. If not given and *f* has a `name` attribute, that is used for *source*; the default is `'<???>'`.

Novo na versão 3.2: Substitui *readfp()*.

read_string(*string*, *source='<string>'*)
Parse configuration data from a string.

Optional argument *source* specifies a context-specific name of the string passed. If not given, `'<string>'` is used. This should commonly be a filesystem path or a URL.

Novo na versão 3.2.

read_dict(*dictionary*, *source='<dict>'*)
Load configuration from any object that provides a dict-like `items()` method. Keys are section names, values are dictionaries with keys and values that should be present in the section. If the used dictionary type preserves order, sections and their keys will be added in order. Values are automatically converted to strings.

Optional argument *source* specifies a context-specific name of the dictionary passed. If not given, `<dict>` is used.

This method can be used to copy state between parsers.

Novo na versão 3.2.

get(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
Get an *option* value for the named *section*. If *vars* is provided, it must be a dictionary. The *option* is looked up in *vars* (if provided), *section*, and in *DEFAULTSECT* in that order. If the key is not found and *fallback* is provided, it is used as a fallback value. `None` can be provided as a *fallback* value.

All the `'%'` interpolations are expanded in the return values, unless the *raw* argument is true. Values for interpolation keys are looked up in the same manner as the option.

Alterado na versão 3.2: Arguments *raw*, *vars* and *fallback* are keyword only to protect users from trying to use the third argument as the *fallback* fallback (especially when using the mapping protocol).

getint(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
A convenience method which coerces the *option* in the specified *section* to an integer. See *get()* for explanation of *raw*, *vars* and *fallback*.

getfloat(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
A convenience method which coerces the *option* in the specified *section* to a floating point number. See *get()* for explanation of *raw*, *vars* and *fallback*.

getboolean(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
A convenience method which coerces the *option* in the specified *section* to a Boolean value. Note that the accepted values for the option are `'1'`, `'yes'`, `'true'`, and `'on'`, which cause this method to return `True`, and `'0'`, `'no'`, `'false'`, and `'off'`, which cause it to return `False`. These string values are checked in a case-insensitive manner. Any other value will cause it to raise *ValueError*. See *get()* for explanation of *raw*, *vars* and *fallback*.

items(*raw=False*, *vars=None*)
items(*section*, *raw=False*, *vars=None*)
When *section* is not given, return a list of *section_name*, *section_proxy* pairs, including DEFAULTSECT.

Otherwise, return a list of *name*, *value* pairs for the options in the given *section*. Optional arguments have the same meaning as for the *get()* method.

set(*section*, *option*, *value*)
If the given section exists, set the given option to the specified value; otherwise raise *NoSectionError*. *option* and *value* must be strings; if not, *TypeError* is raised.

write(*fileobject*, *space_around_delimiters=True*)
Write a representation of the configuration to the specified *file object*, which must be opened in text mode (accepting strings). This representation can be parsed by a future *read()* call. If *space_around_delimiters* is true, delimiters between keys and values are surrounded by spaces.

remove_option(*section*, *option*)
Remove the specified *option* from the specified *section*. If the section does not exist, raise *NoSectionError*. If the option existed to be removed, return *True*; otherwise return *False*.

remove_section(*section*)
Remove the specified *section* from the configuration. If the section in fact existed, return `True`. Otherwise return `False`.

optionxform(*option*)
Transforms the option name *option* as found in an input file or as passed in by client code to the form that should be used in the internal structures. The default implementation returns a lower-case version of *option*; subclasses may override this or client code can set an attribute of this name on instances to affect this behavior.

You don't need to subclass the parser to use this method, you can also set it on an instance, to a function that takes a string argument and returns a string. Setting it to `str`, for example, would make option names case sensitive:

```
cfgparser = ConfigParser()
cfgparser.optionxform = str
```

Note that when reading configuration files, whitespace around the option names is stripped before *optionxform()* is called.

readfp(*fp*, *filename=None*)
Obsoleto desde a versão 3.2: Use *read_file()* instead.

Alterado na versão 3.2: *readfp()* now iterates on *fp* instead of calling `fp.readline()`.

For existing code calling *readfp()* with arguments which don't support iteration, the following generator may be used as a wrapper around the file-like object:

```
def readline_generator(fp):
    line = fp.readline()
```

(continua na próxima página)

(continuação da página anterior)

```
    while line:
        yield line
        line = fp.readline()
```

Instead of `parser.readfp(fp)` use `parser.read_file(readline_generator(fp))`.

`configparser.`**`MAX_INTERPOLATION_DEPTH`**
The maximum depth for recursive interpolation for `get()` when the *raw* parameter is false. This is relevant only when the default *interpolation* is used.

14.2.10 RawConfigParser Objects

class `configparser.`**`RawConfigParser`**(*defaults=None*, *dict_type=collections.OrderedDict*, *allow_no_value=False*, *, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*[, *interpolation*])
Legacy variant of the *ConfigParser*. It has interpolation disabled by default and allows for non-string section names, option names, and values via its unsafe `add_section` and `set` methods, as well as the legacy `defaults=` keyword argument handling.

Nota: Consider using *ConfigParser* instead which checks types of the values to be stored internally. If you don't want interpolation, you can use `ConfigParser(interpolation=None)`.

`add_section`(*section*)
Add a section named *section* to the instance. If a section by the given name already exists, *DuplicateSectionError* is raised. If the *default section* name is passed, *ValueError* is raised.

Type of *section* is not checked which lets users create non-string named sections. This behaviour is unsupported and may cause internal errors.

`set`(*section*, *option*, *value*)
If the given section exists, set the given option to the specified value; otherwise raise *NoSectionError*. While it is possible to use *RawConfigParser* (or *ConfigParser* with *raw* parameters set to true) for *internal* storage of non-string values, full functionality (including interpolation and output to files) can only be achieved using string values.

This method lets users assign non-string values to keys internally. This behaviour is unsupported and will cause errors when attempting to write to a file or get it in non-raw mode. **Use the mapping protocol API** which does not allow such assignments to take place.

14.2.11 Exceções

exception `configparser.`**`Error`**
Base class for all other *configparser* exceptions.

exception `configparser.`**`NoSectionError`**
Exception raised when a specified section is not found.

exception `configparser.`**`DuplicateSectionError`**
Exception raised if `add_section()` is called with the name of a section that is already present or in strict parsers when a section if found more than once in a single input file, string or dictionary.

Novo na versão 3.2: Optional `source` and `lineno` attributes and arguments to `__init__()` were added.

exception configparser.**DuplicateOptionError**
Exception raised by strict parsers if a single option appears twice during reading from a single file, string or dictionary. This catches misspellings and case sensitivity-related errors, e.g. a dictionary may have two keys representing the same case-insensitive configuration key.

exception configparser.**NoOptionError**
Exception raised when a specified option is not found in the specified section.

exception configparser.**InterpolationError**
Base class for exceptions raised when problems occur performing string interpolation.

exception configparser.**InterpolationDepthError**
Exception raised when string interpolation cannot be completed because the number of iterations exceeds *MAX_INTERPOLATION_DEPTH*. Subclass of *InterpolationError*.

exception configparser.**InterpolationMissingOptionError**
Exception raised when an option referenced from a value does not exist. Subclass of *InterpolationError*.

exception configparser.**InterpolationSyntaxError**
Exception raised when the source text into which substitutions are made does not conform to the required syntax. Subclass of *InterpolationError*.

exception configparser.**MissingSectionHeaderError**
Exception raised when attempting to parse a file which has no section headers.

exception configparser.**ParsingError**
Exception raised when errors occur attempting to parse a file.

Alterado na versão 3.2: The `filename` attribute and `__init__()` argument were renamed to `source` for consistency.

14.3 `netrc` — arquivo de processamento netrc

Código fonte: Lib/netrc.py

A classe *netrc* analisa e encapsula o formato do arquivo netrc usado pelo programa Unix **ftp** e outros clientes FTP.

class netrc.**netrc**([*file*])
Uma instância ou instância de subclasse de *netrc* encapsula dados de um arquivo netrc. O argumento de inicialização, se presente, especifica o arquivo a ser analisado. Se nenhum argumento for fornecido, o arquivo `.netrc` no diretório inicial do usuário – conforme determinado por *os.path.expanduser()* – será lido. Caso contrário, uma exceção *FileNotFoundError* será levantada. Os erros de análise levantam *NetrcParseError* com informações de diagnóstico, incluindo o nome do arquivo, o número da linha e o token final. Se nenhum argumento for especificado em um sistema POSIX, a presença de senhas no arquivo `.netrc` levantará um *NetrcParseError* se a propriedade ou as permissões do arquivo forem inseguras (pertencentes a um usuário que não seja o usuário) executando o processo ou acessível para leitura ou gravação por qualquer outro usuário). Isso implementa um comportamento de segurança equivalente ao do ftp e de outros programas que usam `.netrc`.

Alterado na versão 3.4: Adicionada a verificação de permissão POSIX.

Alterado na versão 3.7: *os.path.expanduser()* é usado para encontrar a localização do arquivo `.netrc` quando *file* não é passado como argumento.

exception netrc.**NetrcParseError**
Exceção levantada pela classe *netrc* quando erros sintáticos são encontrados no texto de origem. As instâncias

desta exceção fornecem três atributos interessantes: `msg` é uma explicação textual do erro, `filename` é o nome do arquivo-fonte e `lineno` fornece o número da linha na qual o erro foi encontrado.

14.3.1 Objetos netrc

Uma instância da classe *netrc* tem os seguintes métodos:

`netrc.`**`authenticators`**`(`*host*`)`
Retorna uma 3-tuple `(login, account, password)` dos autenticadores do *host*. Se o arquivo netrc não contém uma entrada para o host dado, retorna a tupla associada com a entrada padrão. Se não houver nenhum host correpondente nem uma entrada padrão estiver disponível, retorna `None`.

`netrc.`**`__repr__`**`()`
Despeja os dados da classe como uma sequência no formato de um arquivo netrc. (Isso descarta os comentários e pode reordenar as entradas.)

Instâncias de *netrc* possuem variáveis de instância públicas:

`netrc.`**`hosts`**
Dicionário mapeando nomes de host para tuplas `(login, conta, senha)`. A entrada *default*, se houver, é representada como um pseudo-host por esse nome.

`netrc.`**`macros`**
Dicionário mapeando nomes de macros para listas de strings.

Nota: As senhas são limitadas a um subconjunto do conjunto de caracteres ASCII. Todas as pontuações ASCII são permitidas nas senhas; no entanto, observe que caracteres em branco e não imprimíveis não são permitidos nas senhas. Essa é uma limitação da maneira como o arquivo .netrc é analisado e pode ser removido no futuro.

14.4 `xdrlib` — Encode and decode XDR data

Código Fonte: Lib/xdrlib.py

The *xdrlib* module supports the External Data Representation Standard as described in **RFC 1014**, written by Sun Microsystems, Inc. June 1987. It supports most of the data types described in the RFC.

The *xdrlib* module defines two classes, one for packing variables into XDR representation, and another for unpacking from XDR representation. There are also two exception classes.

`class` `xdrlib.`**`Packer`**
Packer is the class for packing data into XDR representation. The *Packer* class is instantiated with no arguments.

`class` `xdrlib.`**`Unpacker`**`(`*data*`)`
`Unpacker` is the complementary class which unpacks XDR data values from a string buffer. The input buffer is given as *data*.

Ver também:

RFC 1014 - XDR: External Data Representation Standard This RFC defined the encoding of data which was XDR at the time this module was originally written. It has apparently been obsoleted by **RFC 1832**.

RFC 1832 - XDR: External Data Representation Standard Newer RFC that provides a revised definition of XDR.

14.4.1 Objetos Packer

Packer instances have the following methods:

Packer.**get_buffer**()
: Returns the current pack buffer as a string.

Packer.**reset**()
: Resets the pack buffer to the empty string.

In general, you can pack any of the most common XDR data types by calling the appropriate `pack_type()` method. Each method takes a single argument, the value to pack. The following simple data type packing methods are supported: `pack_uint()`, `pack_int()`, `pack_enum()`, `pack_bool()`, `pack_uhyper()`, and `pack_hyper()`.

Packer.**pack_float**(*value*)
: Packs the single-precision floating point number *value*.

Packer.**pack_double**(*value*)
: Packs the double-precision floating point number *value*.

The following methods support packing strings, bytes, and opaque data:

Packer.**pack_fstring**(*n*, *s*)
: Packs a fixed length string, *s*. *n* is the length of the string but it is *not* packed into the data buffer. The string is padded with null bytes if necessary to guaranteed 4 byte alignment.

Packer.**pack_fopaque**(*n*, *data*)
: Packs a fixed length opaque data stream, similarly to *pack_fstring()*.

Packer.**pack_string**(*s*)
: Packs a variable length string, *s*. The length of the string is first packed as an unsigned integer, then the string data is packed with *pack_fstring()*.

Packer.**pack_opaque**(*data*)
: Packs a variable length opaque data string, similarly to *pack_string()*.

Packer.**pack_bytes**(*bytes*)
: Packs a variable length byte stream, similarly to *pack_string()*.

The following methods support packing arrays and lists:

Packer.**pack_list**(*list*, *pack_item*)
: Packs a *list* of homogeneous items. This method is useful for lists with an indeterminate size; i.e. the size is not available until the entire list has been walked. For each item in the list, an unsigned integer `1` is packed first, followed by the data value from the list. *pack_item* is the function that is called to pack the individual item. At the end of the list, an unsigned integer `0` is packed.

  For example, to pack a list of integers, the code might appear like this:

  ```
  import xdrlib
  p = xdrlib.Packer()
  p.pack_list([1, 2, 3], p.pack_int)
  ```

Packer.**pack_farray**(*n*, *array*, *pack_item*)
: Packs a fixed length list (*array*) of homogeneous items. *n* is the length of the list; it is *not* packed into the buffer, but a *ValueError* exception is raised if `len(array)` is not equal to *n*. As above, *pack_item* is the function used to pack each element.

Packer.**pack_array**(*list*, *pack_item*)
: Packs a variable length *list* of homogeneous items. First, the length of the list is packed as an unsigned integer, then each element is packed as in *pack_farray()* above.

14.4.2 Objetos Unpacker

The *Unpacker* class offers the following methods:

Unpacker.**reset**(*data*)
Resets the string buffer with the given *data*.

Unpacker.**get_position**()
Returns the current unpack position in the data buffer.

Unpacker.**set_position**(*position*)
Sets the data buffer unpack position to *position*. You should be careful about using *get_position()* and *set_position()*.

Unpacker.**get_buffer**()
Returns the current unpack data buffer as a string.

Unpacker.**done**()
Indicates unpack completion. Raises an *Error* exception if all of the data has not been unpacked.

In addition, every data type that can be packed with a *Packer*, can be unpacked with an *Unpacker*. Unpacking methods are of the form `unpack_type()`, and take no arguments. They return the unpacked object.

Unpacker.**unpack_float**()
Unpacks a single-precision floating point number.

Unpacker.**unpack_double**()
Unpacks a double-precision floating point number, similarly to *unpack_float()*.

In addition, the following methods unpack strings, bytes, and opaque data:

Unpacker.**unpack_fstring**(*n*)
Unpacks and returns a fixed length string. *n* is the number of characters expected. Padding with null bytes to guaranteed 4 byte alignment is assumed.

Unpacker.**unpack_fopaque**(*n*)
Unpacks and returns a fixed length opaque data stream, similarly to *unpack_fstring()*.

Unpacker.**unpack_string**()
Unpacks and returns a variable length string. The length of the string is first unpacked as an unsigned integer, then the string data is unpacked with *unpack_fstring()*.

Unpacker.**unpack_opaque**()
Unpacks and returns a variable length opaque data string, similarly to *unpack_string()*.

Unpacker.**unpack_bytes**()
Unpacks and returns a variable length byte stream, similarly to *unpack_string()*.

The following methods support unpacking arrays and lists:

Unpacker.**unpack_list**(*unpack_item*)
Unpacks and returns a list of homogeneous items. The list is unpacked one element at a time by first unpacking an unsigned integer flag. If the flag is `1`, then the item is unpacked and appended to the list. A flag of `0` indicates the end of the list. *unpack_item* is the function that is called to unpack the items.

Unpacker.**unpack_farray**(*n*, *unpack_item*)
Unpacks and returns (as a list) a fixed length array of homogeneous items. *n* is number of list elements to expect in the buffer. As above, *unpack_item* is the function used to unpack each element.

Unpacker.**unpack_array**(*unpack_item*)
Unpacks and returns a variable length *list* of homogeneous items. First, the length of the list is unpacked as an unsigned integer, then each element is unpacked as in *unpack_farray()* above.

14.4.3 Exceções

Exceptions in this module are coded as class instances:

exception xdrlib.**Error**
The base exception class. *Error* has a single public attribute `msg` containing the description of the error.

exception xdrlib.**ConversionError**
Class derived from *Error*. Contains no additional instance variables.

Here is an example of how you would catch one of these exceptions:

```
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError as instance:
    print('packing the double failed:', instance.msg)
```

14.5 plistlib — Generate and parse Mac OS X .plist files

Código Fonte: Lib/plistlib.py

This module provides an interface for reading and writing the "property list" files used mainly by Mac OS X and supports both binary and XML plist files.

The property list (`.plist`) file format is a simple serialization supporting basic object types, like dictionaries, lists, numbers and strings. Usually the top level object is a dictionary.

To write out and to parse a plist file, use the *dump()* and *load()* functions.

To work with plist data in bytes objects, use *dumps()* and *loads()*.

Values can be strings, integers, floats, booleans, tuples, lists, dictionaries (but only with string keys), *Data*, *bytes*, `bytesarray` or *datetime.datetime* objects.

Alterado na versão 3.4: New API, old API deprecated. Support for binary format plists added.

Ver também:

PList manual page Apple's documentation of the file format.

Este módulo define as seguintes funções:

plistlib.**load**(*fp*, *, *fmt=None*, *use_builtin_types=True*, *dict_type=dict*)
Read a plist file. *fp* should be a readable and binary file object. Return the unpacked root object (which usually is a dictionary).

The *fmt* is the format of the file and the following values are valid:

- *None*: Autodetect the file format
- *FMT_XML*: XML file format
- *FMT_BINARY*: Binary plist format

If *use_builtin_types* is true (the default) binary data will be returned as instances of *bytes*, otherwise it is returned as instances of *Data*.

The *dict_type* is the type used for dictionaries that are read from the plist file.

XML data for the *FMT_XML* format is parsed using the Expat parser from *xml.parsers.expat* – see its documentation for possible exceptions on ill-formed XML. Unknown elements will simply be ignored by the plist parser.

The parser for the binary format raises `InvalidFileException` when the file cannot be parsed.

Novo na versão 3.4.

plistlib.**loads**(*data*, *, *fmt=None*, *use_builtin_types=True*, *dict_type=dict*)
Load a plist from a bytes object. See *load()* for an explanation of the keyword arguments.

Novo na versão 3.4.

plistlib.**dump**(*value*, *fp*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*)
Write *value* to a plist file. *Fp* should be a writable, binary file object.

The *fmt* argument specifies the format of the plist file and can be one of the following values:

- *FMT_XML*: XML formatted plist file
- *FMT_BINARY*: Binary formatted plist file

When *sort_keys* is true (the default) the keys for dictionaries will be written to the plist in sorted order, otherwise they will be written in the iteration order of the dictionary.

When *skipkeys* is false (the default) the function raises *TypeError* when a key of a dictionary is not a string, otherwise such keys are skipped.

A *TypeError* will be raised if the object is of an unsupported type or a container that contains objects of unsupported types.

An *OverflowError* will be raised for integer values that cannot be represented in (binary) plist files.

Novo na versão 3.4.

plistlib.**dumps**(*value*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*)
Return *value* as a plist-formatted bytes object. See the documentation for *dump()* for an explanation of the keyword arguments of this function.

Novo na versão 3.4.

The following functions are deprecated:

plistlib.**readPlist**(*pathOrFile*)
Read a plist file. *pathOrFile* may be either a file name or a (readable and binary) file object. Returns the unpacked root object (which usually is a dictionary).

This function calls *load()* to do the actual work, see the documentation of *that function* for an explanation of the keyword arguments.

Obsoleto desde a versão 3.4: Use *load()*.

Alterado na versão 3.7: Dict values in the result are now normal dicts. You no longer can use attribute access to access items of these dictionaries.

plistlib.**writePlist**(*rootObject*, *pathOrFile*)
Write *rootObject* to an XML plist file. *pathOrFile* may be either a file name or a (writable and binary) file object

Obsoleto desde a versão 3.4: Use *dump()* instead.

plistlib.**readPlistFromBytes**(*data*)
Read a plist data from a bytes object. Return the root object.

See *load()* for a description of the keyword arguments.

Obsoleto desde a versão 3.4: Use *loads()* instead.

Alterado na versão 3.7: Dict values in the result are now normal dicts. You no longer can use attribute access to access items of these dictionaries.

plistlib.**writePlistToBytes**(*rootObject*)
Return *rootObject* as an XML plist-formatted bytes object.

Obsoleto desde a versão 3.4: Use *dumps()* instead.

The following classes are available:

class plistlib.**Data**(*data*)
Return a "data" wrapper object around the bytes object *data*. This is used in functions converting from/to plists to represent the <data> type available in plists.

It has one attribute, data, that can be used to retrieve the Python bytes object stored in it.

Obsoleto desde a versão 3.4: Use a *bytes* object instead.

The following constants are available:

plistlib.**FMT_XML**
The XML format for plist files.

Novo na versão 3.4.

plistlib.**FMT_BINARY**
The binary format for plist files

Novo na versão 3.4.

14.5.1 Exemplos

Generating a plist:

```
pl = dict(
    aString = "Doodah",
    aList = ["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict = dict(
        anotherString = "<hello & hi there!>",
        aThirdString = "M\xe4ssig, Ma\xdf",
        aTrueValue = True,
        aFalseValue = False,
    ),
    someData = b"<binary gunk>",
    someMoreData = b"<lots of binary gunk>" * 10,
    aDate = datetime.datetime.fromtimestamp(time.mktime(time.gmtime())),
)
with open(fileName, 'wb') as fp:
    dump(pl, fp)
```

Parsing a plist:

```
with open(fileName, 'rb') as fp:
    pl = load(fp)
print(pl["aKey"])
```

CAPÍTULO 15

Serviços Criptográficos

Os módulos descritos nesse capítulo implementam vários algoritmos de natureza criptográfica. Eles estão disponíveis a critério da instalação. Em sistemas Unix, o módulo `crypt` pode estar disponível também. Eis uma visão geral:

15.1 `hashlib` — Secure hashes and message digests

Código Fonte: Lib/hashlib.py

This module implements a common interface to many different secure hash and message digest algorithms. Included are the FIPS secure hash algorithms SHA1, SHA224, SHA256, SHA384, and SHA512 (defined in FIPS 180-2) as well as RSA's MD5 algorithm (defined in Internet **RFC 1321**). The terms "secure hash" and "message digest" are interchangeable. Older algorithms were called message digests. The modern term is secure hash.

Nota: If you want the adler32 or crc32 hash functions, they are available in the `zlib` module.

Aviso: Some algorithms have known hash collision weaknesses, refer to the "See also" section at the end.

15.1.1 Hash algorithms

There is one constructor method named for each type of *hash*. All return a hash object with the same simple interface. For example: use `sha256()` to create a SHA-256 hash object. You can now feed this object with *bytes-like objects* (normally `bytes`) using the `update()` method. At any point you can ask it for the *digest* of the concatenation of the data fed to it so far using the `digest()` or `hexdigest()` methods.

Nota: For better multithreading performance, the Python *GIL* is released for data larger than 2047 bytes at object creation or on update.

Nota: Feeding string objects into `update()` is not supported, as hashes work on bytes, not on characters.

Constructors for hash algorithms that are always present in this module are `sha1()`, `sha224()`, `sha256()`, `sha384()`, `sha512()`, *`blake2b()`*, and *`blake2s()`*. `md5()` is normally available as well, though it may be missing if you are using a rare "FIPS compliant" build of Python. Additional algorithms may also be available depending upon the OpenSSL library that Python uses on your platform. On most platforms the `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()` are also available.

Novo na versão 3.6: SHA3 (Keccak) and SHAKE constructors `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()`.

Novo na versão 3.6: *`blake2b()`* and *`blake2s()`* were added.

For example, to obtain the digest of the byte string `b'Nobody inspects the spammish repetition'`:

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xdd}Ae\x15\x93\xc5\xfe\\\x00o\xa5u+7\xfd\xdf\xf7\xbcN\x84:\xa6\xaf\x0c\x95\
↪x0fK\x94\x06'
>>> m.digest_size
32
>>> m.block_size
64
```

More condensed:

```
>>> hashlib.sha224(b"Nobody inspects the spammish repetition").hexdigest()
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

hashlib.**new**(*name*[, *data*])

Is a generic constructor that takes the string *name* of the desired algorithm as its first parameter. It also exists to allow access to the above listed hashes as well as any other algorithms that your OpenSSL library may offer. The named constructors are much faster than *`new()`* and should be preferred.

Using *`new()`* with an algorithm provided by OpenSSL:

```
>>> h = hashlib.new('ripemd160')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

Hashlib provides the following constant attributes:

hashlib.**algorithms_guaranteed**

A set containing the names of the hash algorithms guaranteed to be supported by this module on all platforms. Note that 'md5' is in this list despite some upstream vendors offering an odd "FIPS compliant" Python build that excludes it.

Novo na versão 3.2.

hashlib.**algorithms_available**
: A set containing the names of the hash algorithms that are available in the running Python interpreter. These names will be recognized when passed to *new()*. *algorithms_guaranteed* will always be a subset. The same algorithm may appear multiple times in this set under different names (thanks to OpenSSL).

  Novo na versão 3.2.

The following values are provided as constant attributes of the hash objects returned by the constructors:

hash.**digest_size**
: The size of the resulting hash in bytes.

hash.**block_size**
: The internal block size of the hash algorithm in bytes.

A hash object has the following attributes:

hash.**name**
: The canonical name of this hash, always lowercase and always suitable as a parameter to *new()* to create another hash of this type.

  Alterado na versão 3.4: The name attribute has been present in CPython since its inception, but until Python 3.4 was not formally specified, so may not exist on some platforms.

A hash object has the following methods:

hash.**update**(*data*)
: Update the hash object with the *bytes-like object*. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a+b)`.

  Alterado na versão 3.1: The Python GIL is released to allow other threads to run while hash updates on data larger than 2047 bytes is taking place when using hash algorithms supplied by OpenSSL.

hash.**digest**()
: Return the digest of the data passed to the *update()* method so far. This is a bytes object of size *digest_size* which may contain bytes in the whole range from 0 to 255.

hash.**hexdigest**()
: Like *digest()* except the digest is returned as a string object of double length, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

hash.**copy**()
: Return a copy ("clone") of the hash object. This can be used to efficiently compute the digests of data sharing a common initial substring.

15.1.2 SHAKE variable length digests

The `shake_128()` and `shake_256()` algorithms provide variable length digests with length_in_bits//2 up to 128 or 256 bits of security. As such, their digest methods require a length. Maximum length is not limited by the SHAKE algorithm.

shake.**digest**(*length*)
: Return the digest of the data passed to the `update()` method so far. This is a bytes object of size *length* which may contain bytes in the whole range from 0 to 255.

shake.**hexdigest**(*length*)
: Like *digest()* except the digest is returned as a string object of double length, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

15.1.3 Key derivation

Key derivation and key stretching algorithms are designed for secure password hashing. Naive algorithms such as `sha1(password)` are not resistant against brute-force attacks. A good password hashing function must be tunable, slow, and include a salt.

`hashlib.`**`pbkdf2_hmac`**(*hash_name*, *password*, *salt*, *iterations*, *dklen=None*)

The function provides PKCS#5 password-based key derivation function 2. It uses HMAC as pseudorandom function.

The string *hash_name* is the desired name of the hash digest algorithm for HMAC, e.g. 'sha1' or 'sha256'. *password* and *salt* are interpreted as buffers of bytes. Applications and libraries should limit *password* to a sensible length (e.g. 1024). *salt* should be about 16 or more bytes from a proper source, e.g. `os.urandom()`.

The number of *iterations* should be chosen based on the hash algorithm and computing power. As of 2013, at least 100,000 iterations of SHA-256 are suggested.

dklen is the length of the derived key. If *dklen* is `None` then the digest size of the hash algorithm *hash_name* is used, e.g. 64 for SHA-512.

```
>>> import hashlib, binascii
>>> dk = hashlib.pbkdf2_hmac('sha256', b'password', b'salt', 100000)
>>> binascii.hexlify(dk)
b'0394a2ede332c9a13eb82e9b24631604c31df978b4e2f0fbd2c549944f9d79a5'
```

Novo na versão 3.4.

Nota: A fast implementation of *pbkdf2_hmac* is available with OpenSSL. The Python implementation uses an inline version of `hmac`. It is about three times slower and doesn't release the GIL.

`hashlib.`**`scrypt`**(*password*, ***, *salt*, *n*, *r*, *p*, *maxmem=0*, *dklen=64*)

The function provides scrypt password-based key derivation function as defined in **RFC 7914**.

password and *salt* must be *bytes-like objects*. Applications and libraries should limit *password* to a sensible length (e.g. 1024). *salt* should be about 16 or more bytes from a proper source, e.g. `os.urandom()`.

n is the CPU/Memory cost factor, *r* the block size, *p* parallelization factor and *maxmem* limits memory (OpenSSL 1.1.0 defaults to 32 MiB). *dklen* is the length of the derived key.

Availability: OpenSSL 1.1+.

Novo na versão 3.6.

15.1.4 BLAKE2

BLAKE2 is a cryptographic hash function defined in **RFC 7693** that comes in two flavors:

- **BLAKE2b**, optimized for 64-bit platforms and produces digests of any size between 1 and 64 bytes,
- **BLAKE2s**, optimized for 8- to 32-bit platforms and produces digests of any size between 1 and 32 bytes.

BLAKE2 supports **keyed mode** (a faster and simpler replacement for HMAC), **salted hashing**, **personalization**, and **tree hashing**.

Hash objects from this module follow the API of standard library's `hashlib` objects.

Creating hash objects

New hash objects are created by calling constructor functions:

hashlib.**blake2b**(*data=b'', *, digest_size=64, key=b'', salt=b'', person=b'', fanout=1, depth=1, leaf_size=0, node_offset=0, node_depth=0, inner_size=0, last_node=False*)

hashlib.**blake2s**(*data=b'', *, digest_size=32, key=b'', salt=b'', person=b'', fanout=1, depth=1, leaf_size=0, node_offset=0, node_depth=0, inner_size=0, last_node=False*)

These functions return the corresponding hash objects for calculating BLAKE2b or BLAKE2s. They optionally take these general parameters:

- *data*: initial chunk of data to hash, which must be *bytes-like object*. It can be passed only as positional argument.
- *digest_size*: size of output digest in bytes.
- *key*: key for keyed hashing (up to 64 bytes for BLAKE2b, up to 32 bytes for BLAKE2s).
- *salt*: salt for randomized hashing (up to 16 bytes for BLAKE2b, up to 8 bytes for BLAKE2s).
- *person*: personalization string (up to 16 bytes for BLAKE2b, up to 8 bytes for BLAKE2s).

The following table shows limits for general parameters (in bytes):

Hash	digest_size	len(key)	len(salt)	len(person)
BLAKE2b	64	64	16	16
BLAKE2s	32	32	8	8

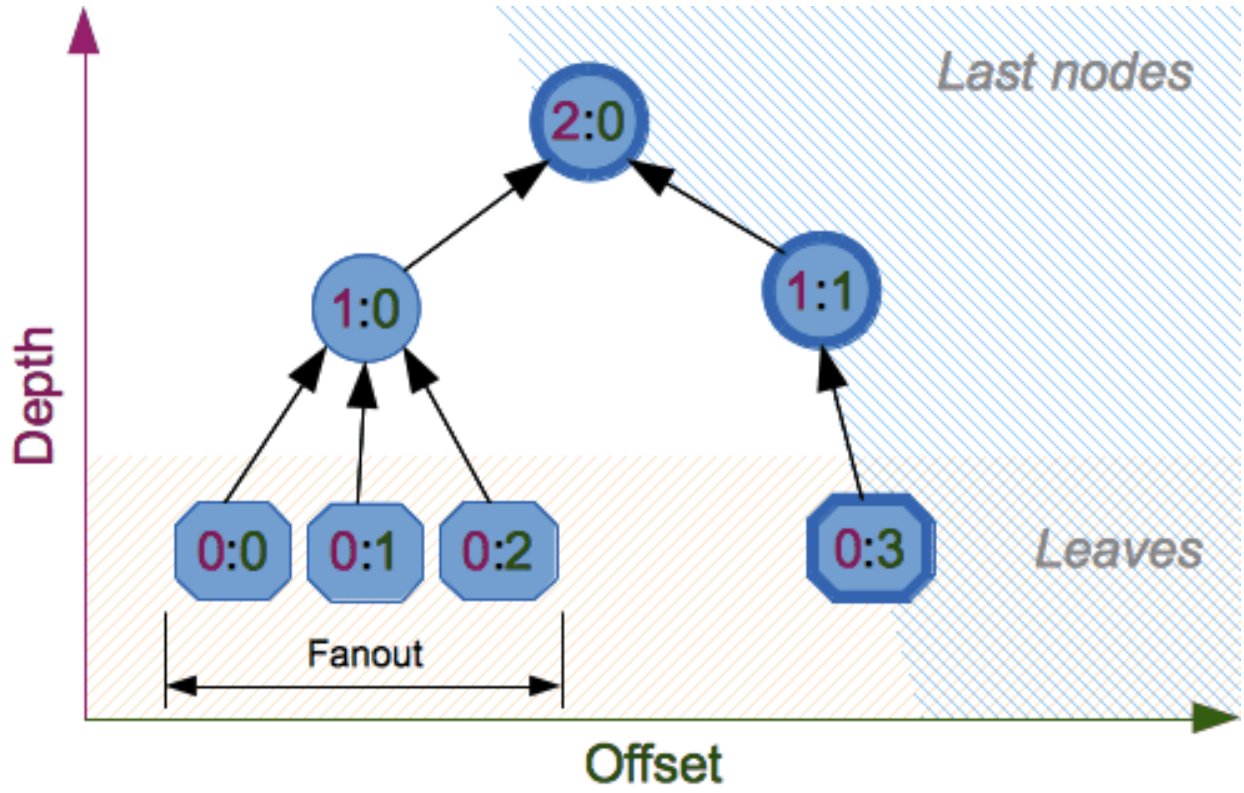
Nota: BLAKE2 specification defines constant lengths for salt and personalization parameters, however, for convenience, this implementation accepts byte strings of any size up to the specified length. If the length of the parameter is less than specified, it is padded with zeros, thus, for example, `b'salt'` and `b'salt\x00'` is the same value. (This is not the case for *key*.)

These sizes are available as module *constants* described below.

Constructor functions also accept the following tree hashing parameters:

- *fanout*: fanout (0 to 255, 0 if unlimited, 1 in sequential mode).
- *depth*: maximal depth of tree (1 to 255, 255 if unlimited, 1 in sequential mode).
- *leaf_size*: maximal byte length of leaf (0 to 2**32-1, 0 if unlimited or in sequential mode).
- *node_offset*: node offset (0 to 2**64-1 for BLAKE2b, 0 to 2**48-1 for BLAKE2s, 0 for the first, leftmost, leaf, or in sequential mode).
- *node_depth*: node depth (0 to 255, 0 for leaves, or in sequential mode).
- *inner_size*: inner digest size (0 to 64 for BLAKE2b, 0 to 32 for BLAKE2s, 0 in sequential mode).
- *last_node*: boolean indicating whether the processed node is the last one (*False* for sequential mode).

See section 2.10 in BLAKE2 specification for comprehensive review of tree hashing.



Constantes

`blake2b.`**`SALT_SIZE`**

`blake2s.`**`SALT_SIZE`**

Salt length (maximum length accepted by constructors).

`blake2b.`**`PERSON_SIZE`**

`blake2s.`**`PERSON_SIZE`**

Personalization string length (maximum length accepted by constructors).

`blake2b.`**`MAX_KEY_SIZE`**

`blake2s.`**`MAX_KEY_SIZE`**

Maximum key size.

`blake2b.`**`MAX_DIGEST_SIZE`**

`blake2s.`**`MAX_DIGEST_SIZE`**

Maximum digest size that the hash function can output.

Exemplos

Simple hashing

To calculate hash of some data, you should first construct a hash object by calling the appropriate constructor function (*blake2b()* or *blake2s()*), then update it with the data by calling `update()` on the object, and, finally, get the digest out of the object by calling `digest()` (or `hexdigest()` for hex-encoded string).

```
>>> from hashlib import blake2b
>>> h = blake2b()
>>> h.update(b'Hello world')
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc
↪'
```

As a shortcut, you can pass the first chunk of data to update directly to the constructor as the positional argument:

```
>>> from hashlib import blake2b
>>> blake2b(b'Hello world').hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc
↪'
```

You can call *hash.update()* as many times as you need to iteratively update the hash:

```
>>> from hashlib import blake2b
>>> items = [b'Hello', b' ', b'world']
>>> h = blake2b()
>>> for item in items:
...     h.update(item)
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc
↪'
```

Using different digest sizes

BLAKE2 has configurable size of digests up to 64 bytes for BLAKE2b and up to 32 bytes for BLAKE2s. For example, to replace SHA-1 with BLAKE2b without changing the size of output, we can tell BLAKE2b to produce 20-byte digests:

```
>>> from hashlib import blake2b
>>> h = blake2b(digest_size=20)
>>> h.update(b'Replacing SHA1 with the more secure function')
>>> h.hexdigest()
'd24f26cf8de66472d58d4e1b1774b4c9158b1f4c'
>>> h.digest_size
20
>>> len(h.digest())
20
```

Hash objects with different digest sizes have completely different outputs (shorter hashes are *not* prefixes of longer hashes); BLAKE2b and BLAKE2s produce different outputs even if the output length is the same:

```
>>> from hashlib import blake2b, blake2s
>>> blake2b(digest_size=10).hexdigest()
'6fa1d8fcfd719046d762'
>>> blake2b(digest_size=11).hexdigest()
'eb6ec15daf9546254f0809'
>>> blake2s(digest_size=10).hexdigest()
'1bf21a98c78a1c376ae9'
>>> blake2s(digest_size=11).hexdigest()
'567004bf96e4a25773ebf4'
```

Keyed hashing

Keyed hashing can be used for authentication as a faster and simpler replacement for Hash-based message authentication code (HMAC). BLAKE2 can be securely used in prefix-MAC mode thanks to the indifferentiability property inherited from BLAKE.

This example shows how to get a (hex-encoded) 128-bit authentication code for message `b'message data'` with key `b'pseudorandom key'`:

```
>>> from hashlib import blake2b
>>> h = blake2b(key=b'pseudorandom key', digest_size=16)
>>> h.update(b'message data')
>>> h.hexdigest()
'3d363ff7401e02026f4a4687d4863ced'
```

As a practical example, a web application can symmetrically sign cookies sent to users and later verify them to make sure they weren't tampered with:

```
>>> from hashlib import blake2b
>>> from hmac import compare_digest
>>>
>>> SECRET_KEY = b'pseudorandomly generated server secret key'
>>> AUTH_SIZE = 16
>>>
>>> def sign(cookie):
...     h = blake2b(digest_size=AUTH_SIZE, key=SECRET_KEY)
...     h.update(cookie)
...     return h.hexdigest().encode('utf-8')
>>>
>>> def verify(cookie, sig):
...     good_sig = sign(cookie)
...     return compare_digest(good_sig, sig)
>>>
>>> cookie = b'user-alice'
>>> sig = sign(cookie)
>>> print("{0},{1}".format(cookie.decode('utf-8'), sig))
user-alice,b'43b3c982cf697e0c5ab22172d1ca7421'
>>> verify(cookie, sig)
True
>>> verify(b'user-bob', sig)
False
>>> verify(cookie, b'0102030405060708090a0b0c0d0e0f00')
False
```

Even though there's a native keyed hashing mode, BLAKE2 can, of course, be used in HMAC construction with *hmac* module:

```
>>> import hmac, hashlib
>>> m = hmac.new(b'secret key', digestmod=hashlib.blake2s)
>>> m.update(b'message')
>>> m.hexdigest()
'e3c8102868d28b5ff85fc35dda07329970d1a01e273c37481326fe0c861c8142'
```

Randomized hashing

By setting *salt* parameter users can introduce randomization to the hash function. Randomized hashing is useful for protecting against collision attacks on the hash function used in digital signatures.

> Randomized hashing is designed for situations where one party, the message preparer, generates all or part of a message to be signed by a second party, the message signer. If the message preparer is able to find cryptographic hash function collisions (i.e., two messages producing the same hash value), then they might prepare meaningful versions of the message that would produce the same hash value and digital signature, but with different results (e.g., transferring $1,000,000 to an account, rather than $10). Cryptographic hash functions have been designed with collision resistance as a major goal, but the current concentration on attacking cryptographic hash functions may result in a given cryptographic hash function providing less collision resistance than expected. Randomized hashing offers the signer additional protection by reducing the likelihood that a preparer can generate two or more messages that ultimately yield the same hash value during the digital signature generation process — even if it is practical to find collisions for the hash function. However, the use of randomized hashing may reduce the amount of security provided by a digital signature when all portions of the message are prepared by the signer.
>
> (NIST SP-800-106 “Randomized Hashing for Digital Signatures”)

In BLAKE2 the salt is processed as a one-time input to the hash function during initialization, rather than as an input to each compression function.

Aviso: *Salted hashing* (or just hashing) with BLAKE2 or any other general-purpose cryptographic hash function, such as SHA-256, is not suitable for hashing passwords. See BLAKE2 FAQ for more information.

```
>>> import os
>>> from hashlib import blake2b
>>> msg = b'some message'
>>> # Calculate the first hash with a random salt.
>>> salt1 = os.urandom(blake2b.SALT_SIZE)
>>> h1 = blake2b(salt=salt1)
>>> h1.update(msg)
>>> # Calculate the second hash with a different random salt.
>>> salt2 = os.urandom(blake2b.SALT_SIZE)
>>> h2 = blake2b(salt=salt2)
>>> h2.update(msg)
>>> # The digests are different.
>>> h1.digest() != h2.digest()
True
```

Personalization

Sometimes it is useful to force hash function to produce different digests for the same input for different purposes. Quoting the authors of the Skein hash function:

> We recommend that all application designers seriously consider doing this; we have seen many protocols where a hash that is computed in one part of the protocol can be used in an entirely different part because two hash computations were done on similar or related data, and the attacker can force the application to make the hash inputs the same. Personalizing each hash function used in the protocol summarily stops this type of attack.
>
> (The Skein Hash Function Family, p. 21)

BLAKE2 can be personalized by passing bytes to the *person* argument:

```
>>> from hashlib import blake2b
>>> FILES_HASH_PERSON = b'MyApp Files Hash'
>>> BLOCK_HASH_PERSON = b'MyApp Block Hash'
>>> h = blake2b(digest_size=32, person=FILES_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'20d9cd024d4fb086aae819a1432dd2466de12947831b75c5a30cf2676095d3b4'
>>> h = blake2b(digest_size=32, person=BLOCK_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'cf68fb5761b9c44e7878bfb2c4c9aea52264a80b75005e65619778de59f383a3'
```

Personalization together with the keyed mode can also be used to derive different keys from a single one.

```
>>> from hashlib import blake2s
>>> from base64 import b64decode, b64encode
>>> orig_key = b64decode(b'Rm5EPJai72qcK3RGBpW3vPNfZy5OZothY+kHY6h21KM=')
>>> enc_key = blake2s(key=orig_key, person=b'kEncrypt').digest()
>>> mac_key = blake2s(key=orig_key, person=b'kMAC').digest()
>>> print(b64encode(enc_key).decode('utf-8'))
rbPb15S/Z9t+agffno5wuhB77VbRi6F9Iv2qIxU7WHw=
>>> print(b64encode(mac_key).decode('utf-8'))
G9GtHFE1YluXY1zWPlYk1e/nWfu0WSEb0KRcjhDeP/o=
```

Modo árvore

Here's an example of hashing a minimal tree with two leaf nodes:

```
  10
 /  \
00  01
```

This example uses 64-byte internal digests, and returns the 32-byte final digest:

```
>>> from hashlib import blake2b
>>>
>>> FANOUT = 2
>>> DEPTH = 2
>>> LEAF_SIZE = 4096
>>> INNER_SIZE = 64
>>>
```

(continua na próxima página)

(continuação da página anterior)

```
>>> buf = bytearray(6000)
>>>
>>> # Left leaf
... h00 = blake2b(buf[0:LEAF_SIZE], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=0, last_node=False)
>>> # Right leaf
... h01 = blake2b(buf[LEAF_SIZE:], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=1, node_depth=0, last_node=True)
>>> # Root node
... h10 = blake2b(digest_size=32, fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=1, last_node=True)
>>> h10.update(h00.digest())
>>> h10.update(h01.digest())
>>> h10.hexdigest()
'3ad2a9b37c6070e374c7a8c508fe20ca86b6ed54e286e93a0318e95e881db5aa'
```

Credits

BLAKE2 was designed by *Jean-Philippe Aumasson*, *Samuel Neves*, *Zooko Wilcox-O'Hearn*, and *Christian Winnerlein* based on SHA-3 finalist BLAKE created by *Jean-Philippe Aumasson*, *Luca Henzen*, *Willi Meier*, and *Raphael C.-W. Phan*.

It uses core algorithm from ChaCha cipher designed by *Daniel J. Bernstein*.

The stdlib implementation is based on pyblake2 module. It was written by *Dmitry Chestnykh* based on C implementation written by *Samuel Neves*. The documentation was copied from pyblake2 and written by *Dmitry Chestnykh*.

The C code was partly rewritten for Python by *Christian Heimes*.

The following public domain dedication applies for both C hash function implementation, extension code, and this documentation:

> To the extent possible under law, the author(s) have dedicated all copyright and related and neighboring rights to this software to the public domain worldwide. This software is distributed without any warranty.
>
> You should have received a copy of the CC0 Public Domain Dedication along with this software. If not, see https://creativecommons.org/publicdomain/zero/1.0/.

The following people have helped with development or contributed their changes to the project and the public domain according to the Creative Commons Public Domain Dedication 1.0 Universal:

- *Alexandr Sokolovskiy*

Ver também:

Module `hmac` A module to generate message authentication codes using hashes.

Módulo `base64` Another way to encode binary hashes for non-binary environments.

https://blake2.net Official BLAKE2 website.

https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf The FIPS 180-2 publication on Secure Hash Algorithms.

https://en.wikipedia.org/wiki/Cryptographic_hash_function#Cryptographic_hash_algorithms Wikipedia article with information on which algorithms have known issues and what that means regarding their use.

https://www.ietf.org/rfc/rfc2898.txt PKCS #5: Password-Based Cryptography Specification Version 2.0

15.2 hmac — Keyed-Hashing for Message Authentication

Código Fonte: Lib/hmac.py

This module implements the HMAC algorithm as described by **RFC 2104**.

hmac.**new**(*key*, *msg=None*, *digestmod=None*)
Return a new hmac object. *key* is a bytes or bytearray object giving the secret key. If *msg* is present, the method call `update(msg)` is made. *digestmod* is the digest name, digest constructor or module for the HMAC object to use. It supports any name suitable to *hashlib.new()* and defaults to the `hashlib.md5` constructor.

Alterado na versão 3.4: Parameter *key* can be a bytes or bytearray object. Parameter *msg* can be of any type supported by *hashlib*. Parameter *digestmod* can be the name of a hash algorithm.

Deprecated since version 3.4, will be removed in version 3.8: MD5 as implicit default digest for *digestmod* is deprecated.

hmac.**digest**(*key*, *msg*, *digest*)
Return digest of *msg* for given secret *key* and *digest*. The function is equivalent to `HMAC(key, msg, digest).digest()`, but uses an optimized C or inline implementation, which is faster for messages that fit into memory. The parameters *key*, *msg*, and *digest* have the same meaning as in *new()*.

CPython implementation detail, the optimized C implementation is only used when *digest* is a string and name of a digest algorithm, which is supported by OpenSSL.

Novo na versão 3.7.

An HMAC object has the following methods:

HMAC.**update**(*msg*)
Update the hmac object with *msg*. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a + b)`.

Alterado na versão 3.4: Parameter *msg* can be of any type supported by *hashlib*.

HMAC.**digest**()
Return the digest of the bytes passed to the *update()* method so far. This bytes object will be the same length as the *digest_size* of the digest given to the constructor. It may contain non-ASCII bytes, including NUL bytes.

> **Aviso:** When comparing the output of *digest()* to an externally-supplied digest during a verification routine, it is recommended to use the *compare_digest()* function instead of the == operator to reduce the vulnerability to timing attacks.

HMAC.**hexdigest**()
Like *digest()* except the digest is returned as a string twice the length containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

> **Aviso:** When comparing the output of *hexdigest()* to an externally-supplied digest during a verification routine, it is recommended to use the *compare_digest()* function instead of the == operator to reduce the vulnerability to timing attacks.

HMAC.**copy**()
Return a copy ("clone") of the hmac object. This can be used to efficiently compute the digests of strings that share a common initial substring.

A hash object has the following attributes:

HMAC.**digest_size**
The size of the resulting HMAC digest in bytes.

HMAC.**block_size**
The internal block size of the hash algorithm in bytes.

Novo na versão 3.4.

HMAC.**name**
The canonical name of this HMAC, always lowercase, e.g. `hmac-md5`.

Novo na versão 3.4.

Este módulo também fornece a seguinte função auxiliar:

hmac.**compare_digest**(*a*, *b*)
Return `a == b`. This function uses an approach designed to prevent timing analysis by avoiding content-based short circuiting behaviour, making it appropriate for cryptography. *a* and *b* must both be of the same type: either *str* (ASCII only, as e.g. returned by *HMAC.hexdigest()*), or a *bytes-like object*.

Nota: If *a* and *b* are of different lengths, or if an error occurs, a timing attack could theoretically reveal information about the types and lengths of *a* and *b*—but not their values.

Novo na versão 3.3.

Ver também:

Módulo *hashlib* The Python module providing secure hash functions.

15.3 `secrets` — Gera números aleatórios seguros para gerenciar segredos

Novo na versão 3.6.

Código Fonte: Lib/secrets.py

O módulo *secrets* é usado para gerar números aleatórios criptograficamente fortes, adequados para o gerenciamento de dados, como senhas, autenticação de conta, tokens de segurança e segredos relacionados.

In particularly, *secrets* should be used in preference to the default pseudo-random number generator in the *random* module, which is designed for modelling and simulation, not security or cryptography.

Ver também:

PEP 506

15.3.1 Números aleatórios

O módulo *secrets* fornece acesso à fonte mais segura de aleatoriedade que seu sistema operacional fornece.

class secrets.**SystemRandom**
Uma classe para gerar números aleatórios usando as fontes da mais alta qualidade fornecidas pelo sistema operacional. Veja *random.SystemRandom* para detalhes adicionais.

secrets.**choice**(*sequence*)
Retorna um elemento escolhido aleatoriamente de uma sequência não vazia.

secrets.**randbelow**(*n*)
Retorna um int aleatório no intervalo [0, *n*).

secrets.**randbits**(*k*)
Retorna um int com *k* bits aleatórios.

15.3.2 Gerando tokens

O módulo *secrets* fornece funções para gerar tokens seguros, adequados para aplicativos como redefinições de senha, URLs difíceis de adivinhar e semelhantes.

secrets.**token_bytes**([*nbytes=None*])
Retorna uma string de byte aleatória contendo *nbytes* número de bytes. Se *nbytes* for `None` ou não fornecido, um padrão razoável é usado.

```
>>> token_bytes(16)
b'\xebr\x17D*t\xae\xd4\xe3S\xb6\xe2\xebP1\x8b'
```

secrets.**token_hex**([*nbytes=None*])
Retorna uma string de texto aleatória, em hexadecimal. A string tem *nbytes* bytes aleatórios, cada byte convertido em dois dígitos hexadecimais. Se *nbytes* for `None` ou não fornecido, um padrão razoável é usado.

```
>>> token_hex(16)
'f9bf78b9a18ce6d46a0cd2b0b86df9da'
```

secrets.**token_urlsafe**([*nbytes=None*])
Retorna uma string de texto segura para URL aleatória, contendo *nbytes* bytes aleatórios. O texto é codificado em Base64, portanto, em média, cada byte resulta em aproximadamente 1,3 caracteres. Se *nbytes* for `None` ou não fornecido, um padrão razoável é usado.

```
>>> token_urlsafe(16)
'Drmhze6EPcv0fN_81Bj-nA'
```

Quantos bytes os tokens devem usar?

Para estar seguro contra ataques de força bruta, os tokens precisam ter aleatoriedade suficiente. Infelizmente, o que é considerado suficiente necessariamente aumentará à medida que os computadores ficarem mais poderosos e capazes de fazer mais suposições em um período mais curto. A partir de 2015, acredita-se que 32 bytes (256 bits) de aleatoriedade são suficientes para o caso de uso típico esperado para o módulo *secrets*.

Para aqueles que desejam gerenciar seu próprio comprimento de token, você pode especificar explicitamente quanta aleatoriedade é usada para tokens, fornecendo um argumento *int* para as várias funções `token_*`. Esse argumento é considerado o número de bytes de aleatoriedade a serem usados.

Caso contrário, se nenhum argumento for fornecido, ou se o argumento for `None`, as funções `token_*` usarão um padrão razoável.

Nota: Esse padrão está sujeito a alterações a qualquer momento, inclusive durante as versões de manutenção.

15.3.3 Outras funções

`secrets.`**`compare_digest`**(*a*, *b*)

Retorna `True` se as strings *a* e *b* forem iguais, caso contrário, `False`, de forma a reduzir o risco de ataques de temporização. Veja *`hmac.compare_digest()`* para detalhes adicionais.

15.3.4 Receitas e melhores práticas

Esta seção mostra as receitas e melhores práticas para usar *`secrets`* para gerenciar um nível básico de segurança.

Gerar uma senha alfanumérica de oito caracteres:

```
import string
alphabet = string.ascii_letters + string.digits
password = ''.join(choice(alphabet) for i in range(8))
```

Nota: Os aplicativos não devem armazenar senhas em um formato recuperável, seja em texto simples ou criptografado. Elas devem ser salgadas e transformadas em hash usando uma função hash de sentido único criptograficamente forte (irreversível).

Gerar uma senha alfanumérica de dez caracteres com pelo menos um caractere minúsculo, pelo menos um caractere maiúsculo e pelo menos três dígitos:

```
import string
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
            and any(c.isupper() for c in password)
            and sum(c.isdigit() for c in password) >= 3):
        break
```

Gerar uma senha longa do estilo XKCD:

```
# On standard Linux systems, use a convenient dictionary file.
# Other platforms may need to provide their own word-list.
with open('/usr/share/dict/words') as f:
    words = [word.strip() for word in f]
    password = ' '.join(choice(words) for i in range(4))
```

Gerar uma URL temporária difícil de adivinhação contendo um token de segurança adequado para aplicativos de recuperação de senha:

```
url = 'https://mydomain.com/reset=' + token_urlsafe()
```

CAPÍTULO 16

Serviços Genéricos do Sistema Operacional

Os módulos descritos neste capítulo fornecem interfaces aos recursos do sistema operacional e que estão disponíveis em (quase) todos os sistemas operacionais, como arquivos e um relógio. As interfaces geralmente são modeladas após as interfaces Unix ou C, mas elas também estão disponíveis na maioria dos outros sistemas. Aqui temos uma visão geral:

16.1 `os` — Diversas interfaces de sistema operacional

Código-fonte: Lib/os.py

Este módulo fornece uma maneira simples de usar funcionalidades que são dependentes de sistema operacional. Se desejar ler ou escrever um arquivo, veja *`open()`*; se o que quer é manipular estruturas de diretórios, veja o módulo *`os.path`*; e se quiser ler todas as linhas, de todos os arquivos, na linha de comando, veja o módulo *`fileinput`*. Para criar arquivos e diretórios temporários, veja o módulo *`tempfile`*; e, para manipulação em alto nível de arquivos e diretórios, veja o módulo *`shutil`*.

Notas sobre a disponibilidade dessas funções:

- O modelo dos módulos dependentes do sistema operacional embutidos no Python é tal que, desde que a mesma funcionalidade esteja disponível, a mesma interface é usada; por exemplo, a função `os.stat(path)` retorna informações estatísticas sobre *path* no mesmo formato (que é originado com a interface POSIX).
- Extensões específicas a um sistema operacional também estão disponíveis através do módulo *`os`*, mas usá-las é, naturalmente, uma ameaça à portabilidade.
- Todas as funções que aceitam nomes de caminhos ou arquivos, aceitam objetos bytes e string, e resultam em um objeto do mesmo tipo, se um caminho ou nome de arquivo for retornado.

Nota: Todas as funções neste módulo trazem *`OSError`* (ou suas subclasses) no caso de nomes e caminhos de arquivos inválidos ou inacessíveis, ou outros argumentos que possuem o tipo correto, mas não são aceitos pelo sistema operacional.

exception os.**error**
Um apelido para a exceção embutida *OSError*.

os.**name**
O nome do módulo importado, dependente do sistema operacional. Atualmente, os seguintes nomes foram registrados: `'posix'`, `'nt'`, `'java'`.

Ver também:

sys.platform tem uma granularidade mais fina. *os.uname()* fornece informação dependentes de versão do sistema.

O módulo *platform* fornece verificações detalhadas sobre a identificação do sistema.

16.1.1 Nomes de arquivos, argumentos de linha de comando e variáveis de ambiente

Em Python, nomes de arquivos, argumentos de linha de comando e variáveis de ambiente são representados usando o tipo string. Em alguns sistemas, é necessária a decodificação dessas sequências de caracteres, de e para bytes, antes de transmiti-las ao sistema operacional. O Python usa a codificação do sistema de arquivos para realizar essa conversão (consulte *sys.getfilesystemencoding()*).

Alterado na versão 3.1: Em alguns sistemas a conversão, usando a codificação do sistema de arquivos, pode falhar. Neste caso, o Python usa o *manipulador de erro de codificação surrogateescape*, o que significa que bytes não decodificados são substituídos por um caractere Unicode U+DCxx na decodificação, e estes são novamente traduzidos para o byte original na codificação.

A codificação do sistema de arquivos deve garantir a decodificação bem-sucedida de todos os bytes abaixo de 128. Se a codificação do sistema de arquivos não fornecer essa garantia, as funções da API poderão gerar UnicodeErrors.

16.1.2 Parâmetros de Processo

Essas funções e itens de dados fornecem informações e operam no processo e usuário atuais.

os.**ctermid**()
Retorna o nome do arquivo correspondente ao terminal de controle do processo.

Disponibilidade: Unix.

os.**environ**
Um objeto de mapeamento representando uma string do ambiente. Por exemplo, `environ['HOME']` é o nome do caminho do seu diretório pessoal (em algumas plataformas), e é equivalente a `getenv("HOME")` em C.

Este mapeamento é capturado na primeira vez que o módulo *os* é importado, normalmente durante a inicialização do Python, como parte do processamento do arquivo `site.py`. Mudanças no ambiente feitas após esse momento não são refletidas em `os.environ`, exceto pelas mudanças feitas modificando `os.environ` diretamente.

Se a plataforma suportar a função *putenv()*, esse mapeamento pode ser usado para modificar o ambiente, além de consultá-lo. *putenv()* será chamado automaticamente quando o mapeamento for modificado.

No Unix, chaves e valores usam *sys.getfilesystemencoding()* e o manipulador de erros `'surrogateescape'`. Use *environb* se quiser usar uma codificação diferente.

Nota: Chamar a função *putenv()* diretamente não muda `os.environ`, por isso é melhor modificar `os.environ`.

Nota: Em algumas plataformas, incluindo FreeBSD e Mac OS X, a modificação de `environ` pode causar vazamentos de memória. Consulte a documentação do sistema para `putenv()`.

Se a função *putenv()* não for fornecida, uma cópia modificada desse mapeamento pode ser passada para as funções apropriadas de criação de processos, para fazer com que os processos filhos usem um ambiente modificado.

Se a plataforma suportar a função *unsetenv()*, os itens nesse mapeamento poderão ser excluídos para remover variáveis de ambiente. *unsetenv()* será chamado automaticamente quando um item é excluído de `os.environ`, e quando um dos métodos `pop()` ou `clear()` é chamado.

`os.`**`environb`**
Versão bytes *environ*: um objeto de mapeamento representando o ambiente como byte strings. *environ* e *environb* são sincronizados (modificar *environb* atualiza *environ*, e vice versa).

environb está disponível somente se *supports_bytes_environ* for `True`.

Novo na versão 3.2.

`os.`**`chdir`**(*path*)
`os.`**`fchdir`**(*fd*)
`os.`**`getcwd`**()
Essas funções são descritas em: *Arquivos e Diretórios*.

`os.`**`fsencode`**(*filename*)
Codifica *filename* como *path-like* para a codificação do sistema de arquivos com o manipulador de erros `'surrogateescape'`, ou `'strict'` no Windows; retorna *bytes* inalterada.

fsdecode() é a função reversa.

Novo na versão 3.2.

Alterado na versão 3.6: Suporte adicionado para aceitar objetos que implementam a interface *os.PathLike*

`os.`**`fsdecode`**(*filename*)
Decodifica *filename* como *path-like* da codificação do sistema de arquivos com o manipulador de erros `'surrogateescape'`, ou `'strict'` no Windows; retorna *str* inalterada.

fsencode() é a função reversa.

Novo na versão 3.2.

Alterado na versão 3.6: Suporte adicionado para aceitar objetos que implementam a interface *os.PathLike*

`os.`**`fspath`**(*path*)
Retorna a representação do sistema de arquivos do caminho.

Se um objeto da classe *str* ou *bytes* é passado, é retornado inalterado. Caso contrário o método *__fspath__()* é chamado, e seu valor é retornado, desde que seja um objeto da classe *str* ou *bytes*. Em todos os outros casos, a exceção *TypeError* é gerada.

Novo na versão 3.6.

`class` `os.`**`PathLike`**
Uma classe base abstrata *abstract base class* para objetos representando um caminho do sistema de arquivos, por exemplo *pathlib.PurePath*.

Novo na versão 3.6.

`abstractmethod` **`__fspath__`**()
Retorna a representação do caminho do sistema de arquivos do objeto.

O método deverá retornar somente objetos *str* ou *bytes*, preferencialmente *str*.

os.**getenv**(*key*, *default=None*)
Retorna o valor da variável de ambiente *key* se existir, ou *default*, se não. *key*, *default* e o resultado são str.

No Unix, chaves e valores são decodificados com a função *sys.getfilesystemencoding()* e o o manipulador de erros `'surrogateescape'`. Use *os.getenvb()* se quiser usar uma codificação diferente.

Disponibilidade: várias versões de Unix, Windows.

os.**getenvb**(*key*, *default=None*)
Retorna o valor da variável de ambiente *key*, se existir, ou *default*, se não existir. *key*, *default* e o resultado são bytes.

getenvb() está disponível somente se *supports_bytes_environ* for `True`.

Disponibilidade: várias versões de Unix.

Novo na versão 3.2.

os.**get_exec_path**(*env=None*)
Retorna a lista de diretórios que serão buscados por um executável nomeado, semelhante a um shell, ao iniciar um processo. *env*, quando especificado, deve ser um dicionário de variáveis de ambiente para procurar o PATH. Por padrão, quando *env* é `None`, *environ* é usado.

Novo na versão 3.2.

os.**getegid**()
Retorna o efetivo ID do grupo do processo atual. Isso corresponde ao bit "set id" no arquivo que está sendo executado no processo atual.

Disponibilidade: Unix.

os.**geteuid**()
Retorna o efetivo ID de usuário do processo atual.

Disponibilidade: Unix.

os.**getgid**()
Retorna o ID real do grupo do processo atual.

Disponibilidade: Unix.

os.**getgrouplist**(*user*, *group*)
Retornar lista de IDs de grupos aos quais *user* pertence. Se *group* não estiver na lista, será incluído; normalmente, *group* é especificado como o campo de ID do grupo, a partir do registro de senha para *user*.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**getgroups**()
Retorna a lista de IDs de grupos suplementares associados ao processo atual.

Disponibilidade: Unix.

Nota: No Mac OS X, o comportamento da função *getgroups()* difere um pouco de outras plataformas Unix. Se o interpretador Python foi compilado para distribuição na versão `10.5` ou anterior, *getgroups()* retorna a lista de ids de grupos efetivos, associados ao processo de usuário atual; esta lista é limitada a um número de entradas definido pelo sistema, tipicamente 16, e pode ser modificada por chamadas para *setgroups()* se tiver o privilégio adequado. Se foi compilado para distribuição na versão maior que `10.5`, *getgroups()* retorna a lista de acesso de grupo atual para o usuário associado ao id de usuário efetivo do processo; a lista de acesso de grupo pode mudar durante a vida útil do processo, e ela não é afetada por chamadas para *setgroups()*, e seu

comprimento não é limitado a 16. O valor da constante MACOSX_DEPLOYMENT_TARGET, pode ser obtido com *sysconfig.get_config_var()*.

---

os.**getlogin**()
Retorna o nome do usuário conectado no terminal de controle do processo. Para a maioria dos propósitos, é mais útil usar *getpass.getuser()*, já que esse último verifica as variáveis de ambiente LOGNAME ou USERNAME para descobrir quem é o usuário, e retorna para pwd.getpwuid(os.getuid())[0] para obter o nome de login do ID do usuário real atual.

Disponibilidade: Unix, Windows.

os.**getpgid**(*pid*)
Retorna o ID do grupo de processo com *pid*. Se *pid* for 0, o ID do grupo do processo atual é retornado.

Disponibilidade: Unix.

os.**getpgrp**()
Retorna o ID do grupo de processo atual.

Disponibilidade: Unix.

os.**getpid**()
Retorna o id do processo atual.

os.**getppid**()
Retorna o ID do processo pai. Quando o processo pai é encerrado, no Unix, o ID retornado é o do processo init(1); no Windows, ainda é o mesmo ID, que já pode ser reutilizado por outro processo.

Disponibilidade: Unix, Windows.

Alterado na versão 3.2: Adicionado suporte para Windows.

os.**getpriority**(*which*, *who*)
Obtém prioridade de agendamento de programa. O valor *which* é um entre *PRIO_PROCESS*, *PRIO_PGRP* ou *PRIO_USER*, e *who* é interpretado em relação a *which* (um identificador de processo para *PRIO_PROCESS*, identificador do grupo de processos para *PRIO_PGRP* e um ID de usuário para *PRIO_USER*). Um valor zero para *who* indica (respectivamente) o processo de chamada, o grupo de processos do processo de chamada ou o ID do usuário real do processo de chamada.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**PRIO_PROCESS**
os.**PRIO_PGRP**
os.**PRIO_USER**
Parâmetros para as funções *getpriority()* e *setpriority()*.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**getresuid**()
Retorna uma tupla (ruid, euid, suid) indicando os IDs de usuário reais, efetivos e salvos do processo atual.

Disponibilidade: Unix.

Novo na versão 3.2.

os.**getresgid**()
Retorna uma tupla (rgid, egid, sgid) indicando os IDs de grupo reais, efetivos e salvos do processo atual.

Disponibilidade: Unix.

Novo na versão 3.2.

os.**getuid**()
Retorna o ID de usuário real do processo atual.

Disponibilidade: Unix.

os.**initgroups**(*username*, *gid*)
Chama o initgroups() do sistema para inicializar a lista de acesso ao grupo com todos os grupos dos quais o nome de usuário especificado é membro, mais o ID do grupo especificado.

Disponibilidade: Unix.

Novo na versão 3.2.

os.**putenv**(*key*, *value*)
Define a variável de ambiente denominada *key* como a string *value*. Tais mudanças no ambiente afetam os subprocessos iniciados com *os.system()*, *popen()* ou *fork()* and *execv()*.

Disponibilidade: várias versões de Unix, Windows.

Nota: Em algumas plataformas, incluindo FreeBSD e Mac OS X, a configuração `environ` pode causar vazamento de memória. Consulte a documentação do sistema para putenv.

Quando há suporte a *putenv()*, as atribuições para itens em `os.environ` são automaticamente convertidas em chamadas correspondentes para *putenv()*; no entanto, chamadas para *putenv()* não atualiza `os.environ`, então é realmente preferível atribuir itens a `os.environ`.

os.**setegid**(*egid*)
Define o ID do grupo efetivo do processo atual.

Disponibilidade: Unix.

os.**seteuid**(*euid*)
Define o ID de usuário efetivo do processo atual.

Disponibilidade: Unix.

os.**setgid**(*gid*)
Define o ID do grupo do processo atual.

Disponibilidade: Unix.

os.**setgroups**(*groups*)
Define a lista de IDs de grupo suplementares associados ao processo atual como *groups*. *groups* deve ser uma sequência e cada elemento deve ser um número inteiro identificando um grupo. Esta operação normalmente está disponível apenas para o superusuário.

Disponibilidade: Unix.

Nota: No Mac OS X, o tamanho de *groups* não pode exceder o número máximo definido pelo sistema de IDs de grupo efetivos, normalmente 16. Consulte a documentação de *getgroups()* para casos em que ele pode não retornar o mesmo conjunto de listas de grupos chamando setgroups().

---

os.**setpgrp**()
Executa a chamada de sistema `setpgrp()` ou `setpgrp (0, 0)` dependendo da versão implementada (se houver). Veja o manual do Unix para a semântica.

Disponibilidade: Unix.

os.**setpgid**(*pid*, *pgrp*)
Faz a chamada de sistema `setpgid()` para definir o ID do grupo do processo com *pid* para o grupo de processos com o id *pgrp*. Veja o manual do Unix para a semântica.

Disponibilidade: Unix.

os.**setpriority**(*which*, *who*, *priority*)
Define a prioridade de agendamento de programa. O valor *which* é um entre *PRIO_PROCESS*, *PRIO_PGRP* ou *PRIO_USER*, e *who* é interpretado em relação a *which* (um identificador de processo para *PRIO_PROCESS*, identificador do grupo de processos para *PRIO_PGRP* e um ID de usuário para *PRIO_USER*). Um valor zero para *who* indica (respectivamente) o processo de chamada, o grupo de processos do processo de chamada ou o ID do usuário real do processo de chamada. *priority* é um valor na faixa de -20 a 19. A prioridade padrão é 0; prioridades menores resultam em agendamento um mais favorável.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**setregid**(*rgid*, *egid*)
Define os IDs de grupo real e efetivo do processo atual

Disponibilidade: Unix.

os.**setresgid**(*rgid*, *egid*, *sgid*)
Define os IDs de grupo real, efetivo e salvo do processo atual

Disponibilidade: Unix.

Novo na versão 3.2.

os.**setresuid**(*ruid*, *euid*, *suid*)
Define os IDs de usuário real, efetivo e salvo do processo atual

Disponibilidade: Unix.

Novo na versão 3.2.

os.**setreuid**(*ruid*, *euid*)
Define os IDs de usuário real e efetivo do processo atual.

Disponibilidade: Unix.

os.**getsid**(*pid*)
Executa a chamada de sistema `getsid()`. Veja o manual do Unix para semântica.

Disponibilidade: Unix.

os.**setsid**()
Executa a chamada de sistema `setsid()`. Veja o manual do Unix para semântica.

Disponibilidade: Unix.

os.**setuid**(*uid*)
Define o ID de usuário do processo atual.

Disponibilidade: Unix.

os.**strerror**(*code*)
Retorna a mensagem de erro correspondente ao código de erro em *code*. Nas plataformas em que `strerror()` retorna `NULL` quando recebe um número de erro desconhecido, *ValueError* é gerada.

os.**supports_bytes_environ**
`True` se o tipo de sistema operacional nativo do ambiente estiver em bytes (ex., `False` no Windows).

Novo na versão 3.2.

os.**umask**(*mask*)
Define o umask numérico atual e retorna o umask anterior.

os.**uname**()
Retorna informações identificando o sistema operacional atual. O valor retornado é um objeto com cinco atributos.

- sysname - nome do sistema operacional
- nodename - nome da máquina na rede (definido pela implementação)
- release - versão do sistema operacional
- version - versão do sistema operacional
- machine - identificador de hardware

Para compatibilidade com versões anteriores, esse objeto também é iterável, comportando-se como uma cinco tupla contendo sysname, nodename, release, version e machine nessa ordem.

Alguns sistemas truncam nodename para 8 caracteres ou para o componente principal; uma maneira melhor de obter o nome do host é *socket.gethostname()* ou até mesmo socket.gethostbyaddr(socket.gethostname()).

Disponibilidade: sabores recentes de Unix.

Alterado na versão 3.3: Retorna tupla alterada de uma tupla para um objeto tipo tupla com atributos nomeados.

os.**unsetenv**(*key*)
Cancela (exclui) a variável de ambiente denominada *key*. Tais mudanças no ambiente afetam subprocessos iniciados com *os.system()*, *popen()* ou *fork()* e *execv()*.

Quando *unsetenv()* é suportado, a exclusão de itens em os.environ é automaticamente traduzida para uma chamada correspondente a *unsetenv()*; no entanto, chamadas a *unsetenv()* não atualizam os.environ, por isso, na verdade é preferível excluir itens de os.environ.

Disponibilidade: várias versões de Unix.

16.1.3 Criação de Objetos Files

This function creates new *file objects*. (See also *open()* for opening file descriptors.)

os.**fdopen**(*fd*, **args*, ***kwargs*)
Retorna um objeto de arquivo aberto conectado ao descritor de arquivo *fd*. Este é um apelido para a função embutida *open()* e aceita os mesmos argumentos. A única diferença é que o primeiro argumento de *fdopen()* deve ser sempre um inteiro.

16.1.4 Operações dos Descritores de Arquivos

Estas funções operam em fluxos de E/S referenciados usando descritores de arquivos.

Descritores de arquivos são pequenos números inteiros correspondentes a um arquivo que foi aberto pelo processo atual. Por exemplo, a entrada padrão geralmente é o descritor de arquivo 0, a saída padrão 1 e erro padrão 2. Outros arquivos abertos por um processo serão então atribuídos 3, 4, 5, e assim por diante. O nome "descritor de arquivo" é um pouco enganoso; em plataformas UNIX, sockets e pipes também são referenciados como descritores de arquivos.

O método *fileno()* pode ser usado para obter o descritor de arquivo associado a um *file object* quando solicitado. Note-se que usar o descritor de arquivo diretamente ignorará os métodos do objeto de arquivo, ignorando aspectos como buffer interno de dados.

os.**close**(*fd*)
Fecha o descritor de arquivo * fd *.

Nota: Esta função destina-se a E/S de baixo nível e deve ser aplicada a um descritor de arquivo retornado por *os.open()* ou *pipe()*. Para fechar um "objeto arquivo" retornado pela função embutida *open()* ou by *popen()* ou *fdopen()*, use seu método *close()*.

os.**closerange**(*fd_low*, *fd_high*)
Feche todos os descritores de arquivo de *fd_low* (inclusivo) a *fd_high* (exclusivo), ignorando erros. Equivalente a (mas muito mais rápido do que):

```
for fd in range(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

os.**device_encoding**(*fd*)
Retorna uma string descrevendo a codificação do dispositivo associado a *fd* se estiver conectado a um terminal; senão retorna *None*.

os.**dup**(*fd*)
Retorna uma cópia do descritor de arquivo *fd*. O novo descritor de arquivo é *non-inheritable*.

No Windows, ao duplicar um fluxo padrão (0: stdin, 1: stdout, 2: stderr), o novo descritor de arquivo é *inheritable*.

Alterado na versão 3.4: O novo descritor de arquivo agora é não-hereditário.

os.**dup2**(*fd*, *fd2*, *inheritable=True*)
Duplica o descritor de arquivo *fd* como *fd2*, fechando o último em primeiro lugar, se necessário. Retorna *fd2*. O novo descritor de arquivo é *inheritable* por padrão ou não-hereditário se *inheritable* for `False`.

Alterado na versão 3.4: Adicionar o parâmetro opcional *inheritable*.

Alterado na versão 3.7: Retorna *fd2* em caso de sucesso. Anteriormente, retornava sempre `None`.

os.**fchmod**(*fd*, *mode*)
Altera o modo do arquivo dado por *fd* ao *mode* numérico. Veja a documentação de *chmod()* para valores possíveis de *mode*. A partir do Python 3.3, isto é equivalente a `os.chmod(fd, mode)`.

Disponibilidade: Unix.

os.**fchown**(*fd*, *uid*, *gid*)
Altera o o ID do proprietário e do grupo do arquivo dado por *fd* para o *uid* e *gid* numérico. Para deixar um dos ids inalteradas, defina-o como -1. Veja *chown()*. A partir do Python 3.3, isto é equivalente a `os.chown(fd, uid, gid)`.

Disponibilidade: Unix.

os.**fdatasync**(*fd*)
Força de gravação de arquivo com descritor de arquivo *fd* no disco. Não força a atualização de metadados.

Disponibilidade: Unix.

Nota: Esta função não está disponível no MacOS.

os.**fpathconf**(*fd*, *name*)
Retorna informações de configuração de sistema relevantes para um arquivo aberto. *name* especifica o valor de

configuração para recuperar; pode ser uma string que é o nome de um valor do sistema definido; estes nomes são especificados em uma série de padrões (POSIX.1, Unix 95, Unix 98 e outros). Algumas plataformas definem nomes adicionais também. Os nomes conhecidos do sistema operacional hospedeiro são dadas no dicionário `pathconf_names`. Para variáveis de configuração não incluídas neste mapeamento, também é aceito passar um número inteiro para *name*.

Se *name* for uma string e não for conhecida, *ValueError* é gerado. Se um valor específico para *name* não for compatível com o sistema de host, mesmo que seja incluído no `pathconf_names`, um erro *OSError* é gerado com *errno.EINVAL* como número do erro.

Assim como no Python 3.3, é equivalente a `os.pathconf(fd, name)`.

Disponibilidade: Unix.

`os.`**`fstat`**(*fd*)
Captura o status do descritor de arquivo *fd*. Retorna um objeto *stat_result*.

Assim como no Python 3.3, é equivalente a `os.stat(fd)`.

Ver também:

A função *stat()*.

`os.`**`fstatvfs`**(*fd*)
Retorna informações sobre o sistema de arquivos que contém o arquivo associado com descritor de arquivo *fd*, como *statvfs()*. A partir do Python 3.3, isso equivale a `os.statvfs(fd)`.

Disponibilidade: Unix.

`os.`**`fsync`**(*fd*)
Gravação à força no disco de arquivo com descritor de arquivo *fd*. No Unix, isto chama a função nativa `fsync()`; no Windows, a função de MS `_commit()`.

Se você estiver começando com buffer no Python *file object* *f*, primeiro use `f.flush()`, e depois use `os.fsync(f.fileno())`, para garantir que todos os buffers internos associados com *f* sejam gravados no disco.

Disponibilidade: Unix, Windows.

`os.`**`ftruncate`**(*fd*, *length*)
Trunca o arquivo correspondente ao descritor de arquivo *fd*, de modo que tenha no máximo *length* bytes de tamanho. A partir do Python 3.3, isto é equivalente a `os.truncate(fd, length)`.

Disponibilidade: Unix, Windows.

Alterado na versão 3.5: Adicionado suporte para o Windows.

`os.`**`get_blocking`**(*fd*)
Obtém o modo de bloqueio do descritor de arquivo: `False` se o flag *O_NONBLOCK* estiver marcado, `True` se o flag estiver desmarcado.

Veja também *set_blocking()* e *socket.socket.setblocking()*.

Disponibilidade: Unix.

Novo na versão 3.5.

`os.`**`isatty`**(*fd*)
Retorna `True` se o descritor de arquivo *fd* estiver aberto e conectado a um dispositivo do tipo tty, senão `False`.

`os.`**`lockf`**(*fd*, *cmd*, *len*)
Aplica, testa ou remove um bloqueio POSIX em um descritor de arquivo aberto. *fd* é um descritor de arquivo aberto. *cmd* especifica o comando a ser usado - um dentre *F_LOCK*, *F_TLOCK*, *F_ULOCK* ou *F_TEST*. *len* especifica a seção do arquivo a ser bloqueada.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**F_LOCK**
os.**F_TLOCK**
os.**F_ULOCK**
os.**F_TEST**
Flags que especificam qual ação *lockf()* vai realizar.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**lseek**(*fd*, *pos*, *how*)
Define a posição atual do descritor de arquivo *fd* para a posição *pos*, modificada por *how*: *SEEK_SET* ou 0 para definir a posição em relação ao início do arquivo; *SEEK_CUR* ou 1 para defini-la em relação à posição atual; *SEEK_END* ou 2 para defini-la em relação ao final do arquivo. Retorna a nova posição do cursor em bytes, a partir do início.

os.**SEEK_SET**
os.**SEEK_CUR**
os.**SEEK_END**
Parâmetros para a function *lseek()*. Seus valores são respectivamente 0, 1, e 2.

Novo na versão 3.3: Alguns sistemas operacionais podem suportar valores adicionais, como os.SEEK_HOLE ou os.SEEK_DATA.

os.**open**(*path*, *flags*, *mode=0o777*, ***, *dir_fd=None*)
Abre o arquivo *path* e define várias flags de acordo com *flags* e, possivelmente, seu modo, de acordo com *mode*. Ao computar *mode*, o valor atual de umask é iniciar com máscara. Retorna o descritor de arquivo para o arquivo recém-aberto. O novo descritor de arquivo é *non-inheritable*.

Para ler uma descrição dos valores de flags e modos, veja a documentação de tempo de execução C; constantes de flag (como *O_RDONLY* e *O_WRONLY*) são definidas no módulo *os*. Em particular, no Windows é necessário adicionar *O_BINARY* para abrir arquivos em modo binário.

Esta função pode suportar *paths relative to directory descriptors* com o parâmetro *dir_fd*.

Alterado na versão 3.4: O novo descritor de arquivo agora é não-hereditário.

Nota: Esta função é destinada a E/S de baixo nível. Para uso normal, use a função embutida *open()*, que retorna um *file object* com os métodos read() e write() (e muitos mais). Para envolver um descritor de arquivo em um objeto de arquivo, use *fdopen()*.

Novo na versão 3.3: O argumento *dir_fd*.

Alterado na versão 3.5: Se a chamada de sistema é interrompida e o tratador de sinal não levanta uma exceção, a função agora tenta novamente a chamada de sistema em vez de levantar uma exceção *InterruptedError* (consulte **PEP 475** para entender a lógica).

Alterado na versão 3.6: Aceita um *path-like object*.

As seguintes constantes são opções para o parâmetro *flags* da função *open()*. Elas podem ser combinadas usando o operador bitwise OR |. Algumas delas não estão disponíveis em todas as plataformas. Para obter descrições de sua disponibilidade e uso, consulte a página do manual *open(2)* no Unix ou the MSDN no Windows.

os.**O_RDONLY**
os.**O_WRONLY**
os.**O_RDWR**
os.**O_APPEND**
os.**O_CREAT**

os.**O_EXCL**
os.**O_TRUNC**
As constantes acima estão disponíveis no Unix e Windows.

os.**O_DSYNC**
os.**O_RSYNC**
os.**O_SYNC**
os.**O_NDELAY**
os.**O_NONBLOCK**
os.**O_NOCTTY**
os.**O_CLOEXEC**
As constantes acima estão disponíveis apenas no Unix.

Alterado na versão 3.3: adiciona a constante *O_CLOEXEC*.

os.**O_BINARY**
os.**O_NOINHERIT**
os.**O_SHORT_LIVED**
os.**O_TEMPORARY**
os.**O_RANDOM**
os.**O_SEQUENTIAL**
os.**O_TEXT**
As constantes acima estão disponíveis apenas no Windows.

os.**O_ASYNC**
os.**O_DIRECT**
os.**O_DIRECTORY**
os.**O_NOFOLLOW**
os.**O_NOATIME**
os.**O_PATH**
os.**O_TMPFILE**
os.**O_SHLOCK**
os.**O_EXLOCK**
As constantes acima são extensões e não estarão presentes, se não forem definidos pela biblioteca C.

Alterado na versão 3.4: Adiciona *O_PATH* em sistemas que suportam. Adiciona *O_TMPFILE*, só está disponível em Linux Kernel 3.11 ou mais recente.

os.**openpty**()
Abrir um novo par de pseudo-terminal. Retorna um par de descritores de arquivos `(master, slave)` para o pty e o tty, respectivamente. Os novos descritores de arquivos são *non-inheritable*. Para uma abordagem (ligeiramente) mais portátil, use o módulo *pty*.

Availability: alguns tipos de Unix.

Alterado na versão 3.4: Os novos descritores de arquivos agora são não-herdáveis.

os.**pipe**()
Cria um encadeamento. Retorna um par de descritores de arquivos `(r, w)` usáveis para leitura e escrita, respectivamente. O novo descritor de arquivo é *non-inheritable*.

Disponibilidade: Unix, Windows.

Alterado na versão 3.4: Os novos descritores de arquivos agora são não-herdáveis.

os.**pipe2**(*flags*)
Cria um canal com *flags* definidos atomicamente. *flags* podem ser construídos por ORing junto a um ou mais destes valores: *O_NONBLOCK*, *O_CLOEXEC*. Retorna um par de descritores de arquivo `(r, w)` utilizáveis para leitura e gravação, respectivamente.

Availability: alguns tipos de Unix.

Novo na versão 3.3.

os.**posix_fallocate**(*fd*, *offset*, *len*)
Garante que espaço em disco suficiente seja alocado para o arquivo especificado por *fd* iniciando em *offset* e continuando por *len* bytes.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**posix_fadvise**(*fd*, *offset*, *len*, *advice*)
Anuncia a intenção de acessar dados em um padrão específico, permitindo assim que o kernel faça otimizações. O aviso se aplica à região do arquivo especificado por *fd*, iniciando em *offset* e continuando por *len* bytes. *advice* é um entre `POSIX_FADV_NORMAL`, `POSIX_FADV_SEQUENTIAL`, `POSIX_FADV_RANDOM`, `POSIX_FADV_NOREUSE`, `POSIX_FADV_WILLNEED` ou `POSIX_FADV_DONTNEED`.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**POSIX_FADV_NORMAL**
os.**POSIX_FADV_SEQUENTIAL**
os.**POSIX_FADV_RANDOM**
os.**POSIX_FADV_NOREUSE**
os.**POSIX_FADV_WILLNEED**
os.**POSIX_FADV_DONTNEED**
Sinalizadores que podem ser usados em *advice* em `posix_fadvise()` que especificam o padrão de acesso que provavelmente será usado.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**pread**(*fd*, *n*, *offset*)
Lê no máximo *n* bytes do descritor de arquivo *fd* na posição *offset*, mantendo o deslocamento do arquivo inalterado.

Retorna uma bytestring contendo os bytes lidos. Se o final do arquivo referido por *fd* for atingido, um objeto de bytes vazios será retornado.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**preadv**(*fd*, *buffers*, *offset*, *flags=0*)
Lê de um descritor de arquivo *fd* na posição de *offset* em *buffers* mutáveis de *objetos do tipo bytes*, deixando o deslocamento do arquivo inalterado. Transfire os dados para cada buffer até ficar cheio e depois passa para o próximo buffer na sequência para armazenar o restante dos dados.

O argumento *flags* contém um OR bit a bit de zero ou mais dos seguintes sinalizadores:

- `RWF_HIPRI`
- `RWF_NOWAIT`

Retorna o número total de bytes realmente lidos, que pode ser menor que a capacidade total de todos os objetos.

O sistema operacional pode definir um limite (`sysconf()` valor `'SC_IOV_MAX'`) no número de buffers que podem ser usados.

Combina a funcionalidade de `os.readv()` e `os.pread()`.

Disponibilidade: Linux 2.6.30 e posterior, FreeBSD 6.0 e posterior, OpenBSD 2.7 e posterior. O use de sinalizadores requer Linux 4.6 ou posterior.

Novo na versão 3.7.

os.**RWF_NOWAIT**
Não aguarda por dados que não estão disponíveis imediatamente. Se esse sinalizador for especificado, a chamada do sistema retorna instantaneamente se for necessário ler dados do armazenamento de backup ou aguarda um bloqueio.

Se alguns dados foram lidos com sucesso, ele retorna o número de bytes lidos. Se nenhum bytes foi lido, ele retornará `-1` e definirá errno como *`errno.EAGAIN`*.

Availability: Linux 4.14 e mais novos.

Novo na versão 3.7.

os.**RWF_HIPRI**
Alta prioridade de leitura/gravação. Permite sistemas de arquivos baseados em blocos para usar a consulta do dispositivo, que fornece latência inferior, mas pode usar recursos adicionais.

Atualmente, no Linux, esse recurso é usável apenas em um descritor de arquivo aberto usando o sinalizador *`O_DIRECT`*.

Availability: Linux 4.6 e mais novos.

Novo na versão 3.7.

os.**pwrite**(*fd*, *str*, *offset*)
Lê no máximo *n* bytes do descritor de arquivo *fd* na posição *offset*, mantendo o deslocamento do arquivo inalterado.

Retorna o número de bytes realmente escritos.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**pwritev**(*fd*, *buffers*, *offset*, *flags=0*)
Escreve o conteúdo de *buffers* no descritor de arquivo *fd* em um deslocamento *offset*, deixando o deslocamento do arquivo inalterado. *buffers* deve ser uma sequência de *objetos bytes ou semelhantes*. Os buffers são processados em ordem de matriz. Todo o conteúdo do primeiro buffer é gravado antes de prosseguir para o segundo, e assim por diante.

O argumento *flags* contém um OR bit a bit de zero ou mais dos seguintes sinalizadores:

- *`RWF_DSYNC`*
- *`RWF_SYNC`*

Retorna o número total de bytes realmente escritos.

O sistema operacional pode definir um limite (*`sysconf()`* valor `'SC_IOV_MAX'`) no número de buffers que podem ser usados.

Combina a funcionalidade de *`os.writev()`* e *`os.pwrite()`*.

Availability: Linux 2.6.30 and newer, FreeBSD 6.0 and newer, OpenBSD 2.7 and newer. Using flags requires Linux 4.7 or newer.

Novo na versão 3.7.

os.**RWF_DSYNC**
Fornece um equivalente por gravação do sinalizador `open(2)` de *`O_DSYNC`*. Este efeito de sinalizador se aplica apenas ao intervalo de dados escrito pela chamada de sistema.

Disponibilidade: Linux 4.7 e mais novos.

Novo na versão 3.7.

os.**RWF_SYNC**
Fornece um equivalente por gravação do sinalizador `open(2)` de *O_SYNC*. Este efeito de sinalizador se aplica apenas ao intervalo de dados escrito pela chamada de sistema.

Disponibilidade: Linux 4.7 e mais novos.

Novo na versão 3.7.

os.**read**(*fd*, *n*)
Lê no máximo *n* bytes do descritor de arquivos *fd*.

Retorna uma bytestring contendo os bytes lidos. Se o final do arquivo referido por *fd* for atingido, um objeto de bytes vazios será retornado.

Nota: Esta função destina-se a E/S de baixo nível e deve ser aplicada a um descritor de arquivo retornado por *os.open()* ou *pipe()*. Para ler um "objeto arquivo" retornado pela função embutida *open()* ou por *popen()* ou *fdopen()*, ou *sys.stdin*, use seus métodos `read()` ou `readline()`.

Alterado na versão 3.5: Se a chamada de sistema é interrompida e o tratador de sinal não levanta uma exceção, a função agora tenta novamente a chamada de sistema em vez de levantar uma exceção *InterruptedError* (consulte **PEP 475** para entender a lógica).

os.**sendfile**(*out*, *in*, *offset*, *count*)
os.**sendfile**(*out*, *in*, *offset*, *count*[, *headers*][, *trailers*], *flags=0*)
Copy *count* bytes from file descriptor *in* to file descriptor *out* starting at *offset*. Return the number of bytes sent. When EOF is reached return 0.

A primeira notação de função é suportada por todas as plataformas que definem *sendfile()*.

On Linux, if *offset* is given as `None`, the bytes are read from the current position of *in* and the position of *in* is updated.

The second case may be used on Mac OS X and FreeBSD where *headers* and *trailers* are arbitrary sequences of buffers that are written before and after the data from *in* is written. It returns the same as the first case.

On Mac OS X and FreeBSD, a value of 0 for *count* specifies to send until the end of *in* is reached.

All platforms support sockets as *out* file descriptor, and some platforms allow other types (e.g. regular file, pipe) as well.

Os aplicativos de plataforma cruzada não devem usar os argumentos *headers*, *trailers* e *flags*.

Disponibilidade: Unix.

Nota: Para um wrapper de nível superior de *sendfile()*, consulte *socket.socket.sendfile()*.

Novo na versão 3.3.

os.**set_blocking**(*fd*, *blocking*)
Define o modo de bloqueio do descritor de arquivo especificado. Define o sinalizador *O_NONBLOCK* se bloqueio for `False`; do contrário, limpa o sinalizador.

Veja também *get_blocking()* e *socket.socket.setblocking()*.

Disponibilidade: Unix.

Novo na versão 3.5.

os.**SF_NODISKIO**
os.**SF_MNOWAIT**

os.**SF_SYNC**
Parâmetros para a função *sendfile()*, se a implementação tiver suporte a eles.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**readv**(*fd*, *buffers*)
Lê de um descritor de arquivo *fd* em um número de *buffers* mutáveis *objetos bytes ou similar*. Transfere os dados para cada buffer até que esteja cheio e, a seguir, vai para o próximo buffer na sequência para armazenar o restante dos dados.

Retorna o número total de bytes realmente lidos, que pode ser menor que a capacidade total de todos os objetos.

O sistema operacional pode definir um limite (*sysconf()* valor 'SC_IOV_MAX') no número de buffers que podem ser usados.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**tcgetpgrp**(*fd*)
Retorna o grupo de processos associado ao terminal fornecido por *fd* (um descritor de arquivo aberto conforme retornado por *os.open()*).

Disponibilidade: Unix.

os.**tcsetpgrp**(*fd*, *pg*)
Define o grupo de processos associado ao terminal fornecido por *fd* (um descritor de arquivo aberto conforme retornado por *os.open()*) para *pg*.

Disponibilidade: Unix.

os.**ttyname**(*fd*)
Retorna uma string que especifica o dispositivo de terminal associado ao descritor de arquivo *fd*. Se *fd* não estiver associado a um dispositivo de terminal, uma exceção é levantada.

Disponibilidade: Unix.

os.**write**(*fd*, *str*)
Escreve o bytestring em *str* no descritor de arquivo *fd*.

Retorna o número de bytes realmente escritos.

Nota: Esta função destina-se a E/S de baixo nível e deve ser aplicada a um descritor de arquivo retornado por *os.open()* ou *pipe()*. Para escrever um "objeto arquivo" retornado pela função embutida *open()* ou por *popen()* ou *fdopen()*, ou *sys.stdout* ou *sys.stderr*, use seu método write().

Alterado na versão 3.5: Se a chamada de sistema é interrompida e o tratador de sinal não levanta uma exceção, a função agora tenta novamente a chamada de sistema em vez de levantar uma exceção *InterruptedError* (consulte **PEP 475** para entender a lógica).

os.**writev**(*fd*, *buffers*)
Escreve o conteúdo de *buffers* no descritor de arquivo *fd*. *buffers* deve ser uma sequência de *objetos bytes ou similar*. Os buffers são processados em ordem de vetor. Todo o conteúdo do primeiro buffer é gravado antes de prosseguir para o segundo, e assim por diante.

Retorna o número total de bytes realmente escritos.

O sistema operacional pode definir um limite (*sysconf()* valor 'SC_IOV_MAX') no número de buffers que podem ser usados.

Disponibilidade: Unix.

Novo na versão 3.3.

Consultando o tamanho de um terminal

Novo na versão 3.3.

os.**get_terminal_size**(*fd=STDOUT_FILENO*)
Retorna o tamanho da janela do terminal como `(columns, lines)`, tupla do tipo *terminal_size*.

O argumento opcional `fd` (padrão `STDOUT_FILENO`, ou saída padrão) especifica qual descritor de arquivo deve ser consultado.

Se o descritor de arquivo não estiver conectado a um terminal, uma *OSError* é levantada.

shutil.get_terminal_size() é a função de alto nível que normalmente deve ser usada, `os.get_terminal_size` é a implementação de baixo nível.

Disponibilidade: Unix, Windows.

class os.**terminal_size**
Uma subclasse de tupla, contendo `(columns, lines)` do tamanho da janela do terminal.

columns
Largura da janela do terminal em caracteres.

lines
Altura da janela do terminal em caracteres.

Herança de descritores de arquivo

Novo na versão 3.4.

Um descritor de arquivo tem um sinalizador "herdável" que indica se o descritor de arquivo pode ser herdado por processos filho. Desde o Python 3.4, os descritores de arquivo criados pelo Python não são herdáveis por padrão.

No UNIX, os descritores de arquivo não herdáveis são fechados em processos filho na execução de um novo programa, outros descritores de arquivo são herdados.

No Windows, identificadores não herdáveis e descritores de arquivo são fechados em processos filho, exceto para fluxos padrão (descritores de arquivo 0, 1 e 2: stdin, stdout e stderr), que são sempre herdados. Usando as funções *spawn**, todos os identificadores herdáveis e todos os descritores de arquivo herdáveis são herdados. Usando o módulo *subprocess*, todos os descritores de arquivo, exceto fluxos padrão, são fechados, e os manipuladores herdáveis são herdados apenas se o parâmetro *close_fds* for `False`.

os.**get_inheritable**(*fd*)
Obtém o sinalizador "herdável" do descritor de arquivo especificado (um booleano).

os.**set_inheritable**(*fd*, *inheritable*)
Define o sinalizador "herdável" do descritor de arquivo especificado.

os.**get_handle_inheritable**(*handle*)
Obtém o sinalizador "herdável" do manipulador especificado (um booleano).

Disponibilidade: Windows.

os.**set_handle_inheritable**(*handle*, *inheritable*)
Define o sinalizador "herdável" do manipulador especificado.

Disponibilidade: Windows.

16.1.5 Arquivos e Diretórios

Em algumas plataformas Unix, muitas dessas funções suportam um ou mais destes recursos:

- **specifying a file descriptor:** For some functions, the *path* argument can be not only a string giving a path name, but also a file descriptor. The function will then operate on the file referred to by the descriptor. (For POSIX systems, Python will call the `f...` version of the function.)

  You can check whether or not *path* can be specified as a file descriptor on your platform using *os.supports_fd*. If it is unavailable, using it will raise a *NotImplementedError*.

  If the function also supports *dir_fd* or *follow_symlinks* arguments, it is an error to specify one of those when supplying *path* as a file descriptor.

- **paths relative to directory descriptors:** If *dir_fd* is not `None`, it should be a file descriptor referring to a directory, and the path to operate on should be relative; path will then be relative to that directory. If the path is absolute, *dir_fd* is ignored. (For POSIX systems, Python will call the `...at` or `f...at` version of the function.)

  You can check whether or not *dir_fd* is supported on your platform using *os.supports_dir_fd*. If it is unavailable, using it will raise a *NotImplementedError*.

- **not following symlinks:** If *follow_symlinks* is `False`, and the last element of the path to operate on is a symbolic link, the function will operate on the symbolic link itself instead of the file the link points to. (For POSIX systems, Python will call the `l...` version of the function.)

  You can check whether or not *follow_symlinks* is supported on your platform using *os.supports_follow_symlinks*. If it is unavailable, using it will raise a *NotImplementedError*.

`os.`**`access`**(*path*, *mode*, *, *dir_fd=None*, *effective_ids=False*, *follow_symlinks=True*)

Usa o uid/gid real para testar o acesso ao *path*. Observe que a maioria das operações usará o uid/gid efetivo, portanto, essa rotina pode ser usada em um ambiente suid/sgid para testar se o usuário de chamada tem o acesso especificado ao *path*. *mode* deve ser *F_OK* para testar a existência de *path*, ou pode ser o OU inclusivo de um ou mais dos *R_OK*, *W_OK*, e *X_OK* para testar as permissões. Retorna *True* se o acesso for permitido, *False* se não for. Veja a página man do Unix *access(2)* para mais informações.

Esta função pode suportar a especificação de *caminhos relativos aos descritores de diretório* e *não seguir os links simbólicos*.

Se *effective_ids* for `True`, *access()* irá realizar suas verificações de acesso usando o uid/gid efetivo ao invés do uid/gid real. *effective_ids* pode não ser compatível com sua plataforma; você pode verificar se está ou não disponível usando :data:'os.supports_effective_ids'. Se não estiver disponível, usá-lo levantará uma *NotImplementedError*.

Nota: Usar *access()* para verificar se um usuário está autorizado a, por exemplo, abrir um arquivo antes de realmente fazer isso usando *open()* cria uma brecha de segurança, porque o usuário pode explorar o curto intervalo de tempo entre a verificação e a abertura do arquivo para manipulá-lo. É preferível usar as técnicas: term:*EAFP*. Por exemplo:

```
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()
return "some default data"
```

É melhor escrito como:

```
try:
    fp = open("myfile")
except PermissionError:
    return "some default data"
```

(continua na próxima página)

(continuação da página anterior)

```
else:
    with fp:
        return fp.read()
```

Nota: As operações de E/S podem falhar mesmo quando `access()` indica que elas teriam sucesso, particularmente para operações em sistemas de arquivos de rede que podem ter semântica de permissões além do modelo de bits de permissão POSIX usual.

Alterado na versão 3.3: Adicionados os parâmetros *dir_fd*, *effective_ids* e *follow_symlinks*.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**F_OK**
os.**R_OK**
os.**W_OK**
os.**X_OK**

Valores a serem passados como o parâmetro *mode* de `access()` para testar a existência, legibilidade, capacidade de escrita e executabilidade de *path*, respectivamente.

os.**chdir**(*path*)

Altera o diretório de trabalho atual para *path*.

Esta função pode ter suporte a *especificar um descritor de arquivo*. O descritor deve fazer referência a um diretório aberto, não a um arquivo aberto.

Esta função pode levantar `OSError` e subclasses como `FileNotFoundError`, `PermissionError` e `NotADirectoryError`.

Novo na versão 3.3: Adicionado suporte para especificar *path* como um descritor de arquivo em algumas plataformas.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**chflags**(*path*, *flags*, *, *follow_symlinks=True*)

Define os sinalizadores de *path* para os *flags* numéricos. *flags* podem assumir uma combinação (OU bit a bit) dos seguintes valores (conforme definido no módulo `stat`):

- `stat.UF_NODUMP`
- `stat.UF_IMMUTABLE`
- `stat.UF_APPEND`
- `stat.UF_OPAQUE`
- `stat.UF_NOUNLINK`
- `stat.UF_COMPRESSED`
- `stat.UF_HIDDEN`
- `stat.SF_ARCHIVED`
- `stat.SF_IMMUTABLE`
- `stat.SF_APPEND`
- `stat.SF_NOUNLINK`
- `stat.SF_SNAPSHOT`

Esta função pode ter suporte a *não seguir links simbólicos*.

Disponibilidade: Unix.

Novo na versão 3.3: O argumento *follow_symlinks*.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**chmod**(*path*, *mode*, *, *dir_fd=None*, *follow_symlinks=True*)

Altera o modo de *path* para o *mode* numérico. *mode* pode assumir um dos seguintes valores (conforme definido no módulo `stat`) ou combinações de OU bit a bit deles:

- *stat.S_ISUID*
- *stat.S_ISGID*
- *stat.S_ENFMT*
- *stat.S_ISVTX*
- *stat.S_IREAD*
- *stat.S_IWRITE*
- *stat.S_IEXEC*
- *stat.S_IRWXU*
- *stat.S_IRUSR*
- *stat.S_IWUSR*
- *stat.S_IXUSR*
- *stat.S_IRWXG*
- *stat.S_IRGRP*
- *stat.S_IWGRP*
- *stat.S_IXGRP*
- *stat.S_IRWXO*
- *stat.S_IROTH*
- *stat.S_IWOTH*
- *stat.S_IXOTH*

Esta função pode ter suporte a *especificar um descritor de arquivo*, *caminhos relativos aos descritores de diretório* e *não seguir os links simbólicos*.

Nota: Embora o Windows tenha suporte ao *chmod()*, você só pode definir o sinalizador de somente leitura do arquivo com ele (através das constantes `stat.S_IWRITE` e `stat.S_IREAD` ou um valor inteiro correspondente). Todos os outros bits são ignorados.

Novo na versão 3.3: Adicionado suporte para especificar *path* como um descritor de arquivo aberto e os argumentos *dir_fd* e *follow_symlinks*.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**chown**(*path*, *uid*, *gid*, *, *dir_fd=None*, *follow_symlinks=True*)

Altera o proprietário e o id de grupo de *path* para *uid* e *gid* numéricos. Para deixar um dos ids inalterado, defina-o como -1.

Esta função pode ter suporte a *especificar um descritor de arquivo*, *caminhos relativos aos descritores de diretório* e *não seguir os links simbólicos*.

Consulte `shutil.chown()` para uma função de alto nível que aceita nomes além de ids numéricos.

Disponibilidade: Unix.

Novo na versão 3.3: Added support for specifying an open file descriptor for *path*, and the *dir_fd* and *follow_symlinks* arguments.

Alterado na versão 3.6: Suporta um *path-like object*.

`os.`**`chroot`**(*path*)
Altere o diretório raiz do processo atual para *path*

Disponibilidade: Unix.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.`**`fchdir`**(*fd*)
Altera o diretório de trabalho atual para o diretório representado pelo descritor de arquivo *fd*. O descritor deve se referir a um diretório aberto, não a um arquivo aberto. No Python 3.3, isso é equivalente a `os.chdir(fd)`.

Disponibilidade: Unix.

`os.`**`getcwd`**()
Retorna uma string representando o diretório de trabalho atual.

`os.`**`getcwdb`**()
Retorna uma bytestring representando o diretório de trabalho atual.

`os.`**`lchflags`**(*path*, *flags*)
Define os sinalizadores de *path* para os *flags* numéricos, como `chflags()`, mas não segue links simbólicos. No Python 3.3, isso é equivalente a `os.chflags(path, flags, follow_symlinks=False)`.

Disponibilidade: Unix.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.`**`lchmod`**(*path*, *mode*)
Altera o modo de *path* para o *mode* numérico. Se o caminho for um link simbólico, isso afetará o link simbólico em vez do destino. Veja a documentação de `chmod()` para valores possíveis de *mode*. No Python 3.3, isso é equivalente a `os.chmod(path, mode, follow_symlinks=False)`.

Disponibilidade: Unix.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.`**`lchown`**(*path*, *uid*, *gid*)
Altera o proprietário e o id de grupo de *path* para *uid* e *gid* numéricos. Esta função não seguirá links simbólicos. No Python 3.3, isso é equivalente a `os.chown(path, uid, gid, follow_symlinks=False)`.

Disponibilidade: Unix.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.`**`link`**(*src*, *dst*, ***, *src_dir_fd=None*, *dst_dir_fd=None*, *follow_symlinks=True*)
Cria um link físico apontando para *src* chamado *dst*.

Esta função pode suportar a especificação de *src_dir_fd* e/ou *dst_dir_fd* para fornecer *caminhos relativos a descritores de diretório* e *não seguir links simbólicos*.

Disponibilidade: Unix, Windows.

Alterado na versão 3.2: Adicionado suporte ao Windows.

Novo na versão 3.3: Adiciona os argumentos *src_dir_fd*, *dst_dir_fd* e *follow_symlinks*.

Alterado na versão 3.6: Aceita o termos a *path-like object* para *src* e *dst*.

os.**listdir**(*path='.'*)
Return a list containing the names of the entries in the directory given by *path*. The list is in arbitrary order, and does not include the special entries `'.'` and `'..'` even if they are present in the directory.

path pode ser um *objeto caminho ou similar*. Se *path* for do tipo `bytes` (direta ou indiretamente por meio da interface `PathLike`), os nomes de arquivo retornados também serão do tipo `bytes`; em todas as outras circunstâncias, eles serão do tipo `str`.

Esta função também pode ter suporte a *especificar um descritor de arquivo*; o descritor de arquivo deve fazer referência a um diretório.

Nota: Para codificar nomes de arquivos `str` para `bytes`, use `fsencode()`.

Ver também:

A função `scandir()` retorna entradas de diretório junto com informações de atributo de arquivo, dando melhor desempenho para muitos casos de uso comuns.

Alterado na versão 3.2: O parâmetro *path* tornou-se opcional.

Novo na versão 3.3: Added support for specifying an open file descriptor for *path*.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**lstat**(*path*, *, *dir_fd=None*)
Executa o equivalente a uma chamada de sistema `lstat()` no caminho fornecido. Semelhante a `stat()`, mas não segue links simbólicos. Retorna um objeto `stat_result`.

Em plataformas que não têm suporte a links simbólicos, este é um alias para `stat()`.

No Python 3.3, isso é equivalente a `os.stat(path, dir_fd=dir_fd, follow_symlinks=False)`.

Esta função também pode ter suporte a *caminhos relativos a descritores de diretório*.

Ver também:

A função `stat()`.

Alterado na versão 3.2: Adicionado suporte para links simbólicos do Windows 6.0 (Vista).

Alterado na versão 3.3: Adicionado o parâmetro *dir_fd*.

Alterado na versão 3.6: Aceita o termos a *path-like object* para *src* e *dst*.

os.**mkdir**(*path*, *mode=0o777*, *, *dir_fd=None*)
Cria um diretório chamado *path* com o modo numérico *mode*.

Se o diretório já existe, `FileExistsError` é levantada.

Em alguns sistemas, *mode* é ignorado. Onde ele é usado, o valor atual do umask é primeiro mascarado. Se bits diferentes dos últimos 9 (ou seja, os últimos 3 dígitos da representação octal do *mode*) são definidos, seu significado depende da plataforma. Em algumas plataformas, eles são ignorados e você deve chamar `chmod()` explicitamente para defini-los.

Esta função também pode ter suporte a *caminhos relativos a descritores de diretório*.

Também é possível criar diretórios temporários; veja a função `tempfile.mkdtemp()` do módulo `tempfile`.

Novo na versão 3.3: O argumento *dir_fd*.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**makedirs**(*name*, *mode=0o777*, *exist_ok=False*)
Função de criação recursiva de diretório. Como `mkdir()`, mas torna todos os diretórios de nível intermediário necessários para conter o diretório folha.

O parâmetro *mode* é passado para `mkdir()` para criar o diretório folha; veja *a descrição do mkdir()* para como é interpretado. Para definir os bits de permissão de arquivo de qualquer diretório pai recém-criado, você pode definir o umask antes de invocar `makedirs()`. Os bits de permissão de arquivo dos diretórios pais existentes não são alterados.

Se *exist_ok* for `False` (o padrão), uma `FileExistsError` é levantada se o diretório alvo já existir.

Nota: `makedirs()` ficará confuso se os elementos do caminho a serem criados incluírem `pardir` (por exemplo, ".." em sistemas UNIX).

Esta função trata os caminhos UNC corretamente.

Novo na versão 3.2: O parâmetro *exist_ok*.

Alterado na versão 3.4.1: Antes do Python 3.4.1, se *exist_ok* fosse `True` e o diretório existisse, `makedirs()` ainda levantaria um erro se *mode* não correspondesse ao modo do diretório existente. Como esse comportamento era impossível de implementar com segurança, ele foi removido no Python 3.4.1. Consulte bpo-21082.

Alterado na versão 3.6: Aceita um *path-like object*.

Alterado na versão 3.7: O argumento *mode* não afeta mais os bits de permissão de arquivo de diretórios de nível intermediário recém-criados.

os.**mkfifo**(*path*, *mode=0o666*, ***, *dir_fd=None*)
Cria um FIFO (um encadeamento nomeado) chamado *path* com o modo numérico *mode*. O valor atual de umask é primeiro mascarado do modo.

Esta função também pode ter suporte a *caminhos relativos a descritores de diretório*.

FIFOs são canais que podem ser acessados como arquivos regulares. FIFOs existem até que sejam excluídos (por exemplo, com `os.unlink()`). Geralmente, os FIFOs são usados como ponto de encontro entre os processos do tipo "cliente" e "servidor": o servidor abre o FIFO para leitura e o cliente para escrita. Observe que `mkfifo()` não abre o FIFO – apenas cria o ponto de encontro.

Disponibilidade: Unix.

Novo na versão 3.3: O argumento *dir_fd*.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**mknod**(*path*, *mode=0o600*, *device=0*, ***, *dir_fd=None*)
Cria um nó de sistema de arquivos (arquivo, arquivo especial de dispositivo ou canal nomeado) chamado *path*. *mode* especifica as permissões de uso e o tipo de nó a ser criado, sendo combinado (OU bit a bit) com um de `stat.S_IFREG`, `stat.S_IFCHR`, `stat.S_IFBLK` , e `stat.S_IFIFO` (essas constantes estão disponíveis em `stat`). Para `stat.S_IFCHR` e `stat.S_IFBLK`, *device* define o arquivo especial do dispositivo recém-criado (provavelmente usando `os.makedev()`), caso contrário, ele será ignorado.

Esta função também pode ter suporte a *caminhos relativos a descritores de diretório*.

Disponibilidade: Unix.

Novo na versão 3.3: O argumento *dir_fd*.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**major**(*device*)
Extrai o número principal de dispositivo de um número bruto de dispositivo (normalmente o campo `st_dev` ou `st_rdev` de `stat`).

os.**minor**(*device*)
Extrai o número secundário de dispositivo de um número bruto de dispositivo (geralmente o campo `st_dev` ou `st_rdev` de `stat`).

os.**makedev**(*major*, *minor*)
Compõe um número de dispositivo bruto a partir dos números de dispositivo principais e secundários.

os.**pathconf**(*path*, *name*)
Retorna informações de configuração do sistema relevantes para um arquivo nomeado. *name* especifica o valor de configuração a ser recuperado; pode ser uma string que é o nome de um valor de sistema definido; esses nomes são especificados em vários padrões (POSIX.1, Unix 95, Unix 98 e outros). Algumas plataformas também definem nomes adicionais. Os nomes conhecidos do sistema operacional do host são fornecidos no dicionário `pathconf_names`. Para variáveis de configuração não incluídas nesse mapeamento, passar um número inteiro para *name* também é aceito.

Se *name* for uma string e não for conhecida, `ValueError` é gerado. Se um valor específico para *name* não for compatível com o sistema de host, mesmo que seja incluído no `pathconf_names`, um erro `OSError` é gerado com `errno.EINVAL` como número do erro.

Esta função tem suporte a *especificar um descritor de arquivo*.

Disponibilidade: Unix.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**pathconf_names**
Nomes de mapeamento de dicionário aceitos por `pathconf()` e `fpathconf()` para os valores inteiros definidos para esses nomes pelo sistema operacional do host. Isso pode ser usado para determinar o conjunto de nomes conhecidos pelo sistema.

Disponibilidade: Unix.

os.**readlink**(*path*, *, *dir_fd=None*)
Retorna uma string representando o caminho para o qual o link simbólico aponta. O resultado pode ser um nome de caminho absoluto ou relativo; se for relativo, pode ser convertido para um caminho absoluto usando `os.path.join(os.path.dirname(path), result)`.

Se o *path* for um objeto string (direta ou indiretamente por meio de uma interface `PathLike`), o resultado também será um objeto string e a chamada pode levantará um UnicodeDecodeError. Se o *path* for um objeto de bytes (direto ou indireto), o resultado será um objeto de bytes.

Esta função também pode ter suporte a *caminhos relativos a descritores de diretório*.

Disponibilidade: Unix, Windows.

Alterado na versão 3.2: Adicionado suporte para links simbólicos do Windows 6.0 (Vista).

Novo na versão 3.3: O argumento *dir_fd*.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**remove**(*path*, *, *dir_fd=None*)
Remove (exclui) o arquivo *path*. Se *path* for um diretório, uma `IsADirectoryError` é levantada. Use `rmdir()` para remover diretórios.

Esta função pode suportar *paths relative to directory descriptors*.

No Windows, a tentativa de remover um arquivo que está em uso causa o surgimento de uma exceção; no Unix, a entrada do diretório é removida, mas o armazenamento alocado para o arquivo não é disponibilizado até que o arquivo original não esteja mais em uso.

Esta função é semanticamente idêntica a `unlink()`.

Novo na versão 3.3: O argumento *dir_fd*.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**removedirs**(*name*)
Remove os diretórios recursivamente. Funciona como `rmdir()`, exceto que, se o diretório folha for removido com sucesso, `removedirs()` tenta remover sucessivamente todos os diretórios pai mencionados em *path* até que um erro seja levantado (que é ignorado, porque geralmente significa que um diretório pai não está vazio). Por exemplo, `os.removedirs('foo/bar/baz')` primeiro removerá o diretório `'foo/bar/baz'`, e então removerá `'foo/bar'` e `'foo'` se estiverem vazios. Levanta `OSError` se o diretório folha não pôde ser removido com sucesso.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**rename**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)
Renomeia o arquivo ou diretório *src* para *dst*. Se *dst* existir, a operação falhará com uma subclasse `OSError` em vários casos:

No Windows, se *dst* existir, uma `FileExistsError` é sempre levantada.

No Unix, se *src* é um arquivo e *dst* é um diretório ou vice-versa, uma `IsADirectoryError` ou uma `NotADirectoryError` será levantada respectivamente. Se ambos forem diretórios e *dst* estiver vazio, *dst* será substituído silenciosamente. Se *dst* for um diretório não vazio, uma `OSError` é levantada. Se ambos forem arquivos, *dst* será substituído silenciosamente se o usuário tiver permissão. A operação pode falhar em alguns tipos de Unix se *src* e *dst* estiverem em sistemas de arquivos diferentes. Se for bem-sucedido, a renomeação será uma operação atômica (este é um requisito POSIX).

Esta função pode suportar a especificação de *src_dir_fd* e/ou *dst_dir_fd* para fornecer *caminhos relativos a descritores de diretório*.

Se você quiser sobrescrita multiplataforma do destino, use `replace()`.

Novo na versão 3.3: Os argumentos *src_dir_fd* e *dst_dir_fd*.

Alterado na versão 3.6: Aceita o termos a *path-like object* para *src* e *dst*.

os.**renames**(*old*, *new*)
Diretório recursivo ou função de renomeação de arquivo. Funciona como `rename()`, exceto que a criação de qualquer diretório intermediário necessário para tornar o novo nome de caminho bom é tentada primeiro. Após a renomeação, os diretórios correspondentes aos segmentos de caminho mais à direita do nome antigo serão removidos usando `removedirs()`.

Nota: Esta função pode falhar com a nova estrutura de diretório criada se você não tiver as permissões necessárias para remover o arquivo ou diretório folha.

Alterado na versão 3.6: Aceita um *objeto caminho ou similar* para *old* e *new*.

os.**replace**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)
Renomeia o arquivo ou diretório *src* para *dst*. Se *dst* for um diretório, `OSError` será levantada. Se *dst* existir e for um arquivo, ele será substituído silenciosamente se o usuário tiver permissão. A operação pode falhar se *src* e *dst* estiverem em sistemas de arquivos diferentes. Se for bem-sucedido, a renomeação será uma operação atômica (este é um requisito POSIX).

Esta função pode suportar a especificação de *src_dir_fd* e/ou *dst_dir_fd* para fornecer *caminhos relativos a descritores de diretório*.

Novo na versão 3.3.

Alterado na versão 3.6: Aceita o termos a *path-like object* para *src* e *dst*.

os.**rmdir**(*path*, *, *dir_fd=None*)
Remove (exclui) o diretório *path*. Se o diretório não existe ou não está vazio, uma `FileNotFoundError` ou

uma *OSError* é levantada respectivamente. Para remover árvores de diretório inteiras, pode ser usada *shutil.rmtree()*.

Esta função pode suportar *paths relative to directory descriptors*.

Novo na versão 3.3: O parâmetro *dir_fd*.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**scandir**(*path='.'*)

Return an iterator of *os.DirEntry* objects corresponding to the entries in the directory given by *path*. The entries are yielded in arbitrary order, and the special entries `'.'` and `'..'` are not included.

Usar *scandir()* em vez de *listdir()* pode aumentar significativamente o desempenho do código que também precisa de tipo de arquivo ou informações de atributo de arquivo, porque os objetos *os.DirEntry* expõem essas informações se o sistema operacional fornecer ao digitalizar um diretório. Todos os métodos de *os.DirEntry* podem realizar uma chamada de sistema, mas *is_dir()* e *is_file()* normalmente requerem apenas uma chamada de sistema para links simbólicos; *os.DirEntry.stat()* sempre requer uma chamada de sistema no Unix, mas requer apenas uma para links simbólicos no Windows.

path pode ser um *objeto caminho ou similar*. Se *path* for do tipo `bytes` (direta ou indiretamente por meio da interface *PathLike*), o tipo do atributo *name* e *path* de cada *os.DirEntry* serão `bytes`; em todas as outras circunstâncias, eles serão do tipo `str`.

Esta função também pode ter suporte a *especificar um descritor de arquivo*; o descritor de arquivo deve fazer referência a um diretório.

O iterador *scandir()* suporta o protocolo *gerenciador de contexto* e tem o seguinte método:

scandir.**close**()

Fecha o iterador e libera os recursos adquiridos.

Isso é chamado automaticamente quando o iterador se esgota ou é coletado como lixo, ou quando ocorre um erro durante a iteração. No entanto, é aconselhável chamá-lo explicitamente ou usar a instrução `with`.

Novo na versão 3.6.

O exemplo a seguir mostra um uso simples de *scandir()* para exibir todos os arquivos (excluindo diretórios) no *path* fornecido que não começa com `'.'`. A chamada `entry.is_file()` geralmente não fará uma chamada de sistema adicional:

```
with os.scandir(path) as it:
    for entry in it:
        if not entry.name.startswith('.') and entry.is_file():
            print(entry.name)
```

Nota: Em sistemas baseados no Unix, *scandir()* utilize o sistema opendir() e as funções readdir(). No Windows, será utilizado o Win32 FindFirstFileW e a função FindNextFileW.

Novo na versão 3.5.

Novo na versão 3.6: Adicionado suporte para o protocolo *gerenciador de contexto* e o método *close()*. Se um iterador *scandir()* não estiver esgotado nem explicitamente fechado, uma *ResourceWarning* será emitida em seu destruidor.

A função aceita um *objeto caminho ou similar*.

Alterado na versão 3.7: Adicionado suporte para *descritores de arquivo* no Unix.

class os.**DirEntry**

Objeto produzido por *scandir()* para expor o caminho do arquivo e outros atributos de arquivo de uma entrada de diretório.

scandir() fornecerá o máximo possível dessas informações sem fazer chamadas de sistema adicionais. Quando uma chamada de sistema `stat()` ou `lstat()` é feita, o objeto `os.DirEntry` irá armazenar o resultado em cache.

As instâncias de `os.DirEntry` não devem ser armazenadas em estruturas de dados de longa duração; se você sabe que os metadados do arquivo foram alterados ou se passou muito tempo desde a chamada de *scandir()*, chame `os.stat(entry.path)` para obter informações atualizadas.

Como os métodos `os.DirEntry` podem fazer chamadas ao sistema operacional, eles também podem levantar *OSError*. Se você precisa de um controle muito refinado sobre os erros, você pode pegar *OSError* ao chamar um dos métodos `os.DirEntry` e manipular conforme apropriado.

Para ser diretamente utilizável como um *objeto caminho ou similar*, `os.DirEntry` implementa a interface *PathLike*.

Atributos e métodos em uma instância de `os.DirEntry` são os seguintes:

name

O nome do arquivo base da entrada, relativo ao argumento *path* de *scandir()*.

O atributo *name* será `bytes` se o argumento *path* de *scandir()* for do tipo `bytes` e, caso contrário, `str`. Usa *fsdecode()* para decodificar nomes de arquivos de bytes.

path

O nome do caminho completo da entrada: equivalente a `os.path.join(scandir_path, entry.name)` onde *scandir_path* é o argumento *path* de *scandir()*. O caminho só é absoluto se o argumento *path* de *scandir()* for absoluto. Se o argumento *path* de *scandir()* era um *descritor de arquivo*, o atributo *path* é o mesmo que o atributo *name*.

O atributo *path* será `bytes` se o argumento *path* de *scandir()* for do tipo `bytes` e, caso contrário, `str`. Usa *fsdecode()* para decodificar nomes de arquivos de bytes.

inode()

Retorna o número de nó-i da entrada.

O resultado é armazenado em cache no objeto `os.DirEntry`. Use `os.stat(entry.path, follow_symlinks=False).st_ino` para obter informações atualizadas.

Na primeira chamada sem cache, uma chamada de sistema é necessária no Windows, mas não no Unix.

is_dir(*, *follow_symlinks=True*)

Retorna `True` se esta entrada for um diretório ou um link simbólico apontando para um diretório; retorna `False` se a entrada é ou aponta para qualquer outro tipo de arquivo, ou se ele não existe mais.

Se *follow_symlinks* for `False`, retorna `True` apenas se esta entrada for um diretório (sem seguir os links simbólicos); retorna `False` se a entrada for qualquer outro tipo de arquivo ou se ele não existe mais.

O resultado é armazenado em cache no objeto `os.DirEntry`, com um cache separado para *follow_symlinks* `True` e `False`. Chama *os.stat()* junto com *stat.S_ISDIR()* para buscar informações atualizadas.

Na primeira chamada sem cache, nenhuma chamada do sistema é necessária na maioria dos casos. Especificamente, para links não simbólicos, nem o Windows nem o Unix requerem uma chamada de sistema, exceto em certos sistemas de arquivos Unix, como sistemas de arquivos de rede, que retornam `dirent.d_type == DT_UNKNOWN`. Se a entrada for um link simbólico, uma chamada de sistema será necessária para seguir o link simbólico, a menos que *follow_symlinks* seja `False`.

Este método pode levantar *OSError*, tal como *PermissionError*, mas *FileNotFoundError* é capturada e não levantada.

is_file(*, *follow_symlinks=True*)
Retorna `True` se esta entrada for um arquivo ou um link simbólico apontando para um arquivo; retorna `False` se a entrada é ou aponta para um diretório ou outra entrada que não seja de arquivo, ou se ela não existe mais.

Se *follow_symlinks* for `False`, retorna `True` apenas se esta entrada for um arquivo (sem seguir os links simbólicos); retorna `False` se a entrada for um diretório ou outra entrada que não seja um arquivo, ou se ela não existir mais.

O resultado é armazenado em cache no objeto `os.DirEntry`. Chamadas de sistema em cache feitas e exceções levantadas são conforme *is_dir()*.

is_symlink()
Retorna `True` se esta entrada for um link simbólico (mesmo se quebrado); retorna `False` se a entrada apontar para um diretório ou qualquer tipo de arquivo, ou se ele não existir mais.

O resultado é armazenado em cache no objeto `os.DirEntry`. Chama *os.path.islink()* para buscar informações atualizadas.

Na primeira chamada sem cache, nenhuma chamada do sistema é necessária na maioria dos casos. Especificamente, nem o Windows nem o Unix exigem uma chamada de sistema, exceto em certos sistemas de arquivos Unix, como sistemas de arquivos de rede, que retornam `dirent.d_type == DT_UNKNOWN`.

Este método pode levantar *OSError*, tal como *PermissionError*, mas *FileNotFoundError* é capturada e não levantada.

stat(*, *follow_symlinks=True*)
Retorna um objeto *stat_result* para esta entrada. Este método segue links simbólicos por padrão; para estabelecer um link simbólico, adicione o argumento `follow_symlinks=False`.

On Unix, this method always requires a system call. On Windows, it only requires a system call if *follow_symlinks* is `True` and the entry is a symbolic link.

No Windows, os atributos `st_ino`, `st_dev` e `st_nlink` da *stat_result* são sempre definidos como zero. Chame *os.stat()* para obter esses atributos.

O resultado é armazenado em cache no objeto `os.DirEntry`, com um cache separado para *follow_symlinks* `True` e `False`. Chama *os.stat()* para buscar informações atualizadas.

Note que há uma boa correspondência entre vários atributos e métodos de `os.DirEntry` e de *pathlib.Path*. Em particular, o atributo `name` tem o mesmo significado, assim como os métodos `is_dir()`, `is_file()`, `is_symlink()` e `stat()`.

Novo na versão 3.5.

Alterado na versão 3.6: Adicionado suporte para a interface *PathLike*. Adicionado suporte para caminhos *bytes* no Windows.

os.**stat**(*path*, *, *dir_fd=None*, *follow_symlinks=True*)
Obtém o status de um arquivo ou um descritor de arquivo. Executa o equivalente a uma chamada de sistema `stat()` no caminho fornecido. *path* pode ser especificado como uma string ou bytes – direta ou indiretamente através da interface *PathLike* – ou como um descritor de arquivo aberto. Retorna um objeto *stat_result*.

Esta função normalmente segue links simbólicos; para efetuar stat em um link simbólico, adicione o argumento `follow_symlinks=False`, ou use *lstat()*.

Esta função tem suporte a *especificar um descritor de arquivo* e *não seguir links simbólicos*.

Exemplo:

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
```

(continua na próxima página)

(continuação da página anterior)

```
>>> statinfo
os.stat_result(st_mode=33188, st_ino=7876932, st_dev=234881026,
st_nlink=1, st_uid=501, st_gid=501, st_size=264, st_atime=1297230295,
st_mtime=1297230027, st_ctime=1297230027)
>>> statinfo.st_size
264
```

Ver também:

As funções *fstat()* e *lstat()*.

Novo na versão 3.3: Adicionados os argumentos *dir_fd* e *follow_symlinks*, especificando um descritor de arquivo em vez de um caminho.

Alterado na versão 3.6: Aceita um *path-like object*.

class os.**stat_result**

Objeto cujos atributos correspondem aproximadamente aos membros da estrutura `stat`. É usado para o resultado de *os.stat()*, *os.fstat()* e *os.lstat()*.

Atributos:

st_mode
Modo de arquivo: tipo de arquivo e bits de modo de arquivo (permissões).

st_ino
Dependente da plataforma, mas se diferente de zero, identifica exclusivamente o arquivo para um determinado valor de `st_dev`. Tipicamente:

- o número do nó-i no Unix,
- o índice de arquivo no Windows

st_dev
Identificador do dispositivo no qual este arquivo reside.

st_nlink
Número de links físicos.

st_uid
Identificador de usuário do proprietário do arquivo.

st_gid
Identificador de grupo do proprietário do arquivo.

st_size
Tamanho do arquivo em bytes, se for um arquivo normal ou um link simbólico. O tamanho de um link simbólico é o comprimento do nome do caminho que ele contém, sem um byte nulo final.

Timestamps:

st_atime
Tempo do acesso mais recente expresso em segundos.

st_mtime
Tempo da modificação de conteúdo mais recente expresso em segundos.

st_ctime
Dependente da plataforma:

- a hora da mudança de metadados mais recente no Unix,
- a hora de criação no Windows, expresso em segundos.

st_atime_ns
Hora do acesso mais recente expresso em nanossegundos como um número inteiro.

st_mtime_ns
Hora da modificação de conteúdo mais recente expressa em nanossegundos como um número inteiro.

st_ctime_ns
Dependente da plataforma:

- a hora da mudança de metadados mais recente no Unix,
- a hora de criação no Windows, expresso em nanossegundos como um número inteiro.

Nota: O significado e resolução exatos dos atributos *st_atime*, *st_mtime*, and *st_ctime* dependem do sistema operacional e do sistema de arquivos. Por exemplo, em sistemas Windows que usam os sistemas de arquivos FAT ou FAT32, *st_mtime* tem resolução de 2 segundos, e *st_atime* tem resolução de apenas 1 dia. Consulte a documentação do sistema operacional para obter detalhes.

Da mesma forma, embora *st_atime_ns*, *st_mtime_ns* e *st_ctime_ns* sejam sempre expressos em nanossegundos, muitos sistemas não fornecem precisão de nanossegundos. Em sistemas que fornecem precisão de nanossegundos, o objeto de ponto flutuante usado para armazenar *st_atime*, *st_mtime* e *st_ctime* não pode preservar tudo, e como tal será ligeiramente inexato . Se você precisa dos carimbos de data/hora exatos, sempre deve usar *st_atime_ns*, *st_mtime_ns* e *st_ctime_ns*.

Em alguns sistemas Unix (como Linux), os seguintes atributos também podem estar disponíveis:

st_blocks
Número de blocos de 512 bytes alocados para o arquivo. Isso pode ser menor que *st_size*/512 quando o arquivo possuir lacunas.

st_blksize
Tamanho de bloco "preferido" para E/S eficiente do sistema de arquivos. Gravar em um arquivo em partes menores pode causar uma leitura-modificação-reescrita ineficiente.

st_rdev
Tipo de dispositivo, se for um dispositivo nó-i.

st_flags
Sinalizadores definidos pelo usuário para arquivo.

Em outros sistemas Unix (como o FreeBSD), os seguintes atributos podem estar disponíveis (mas só podem ser preenchidos se o root tentar usá-los):

st_gen
Número de geração do arquivo.

st_birthtime
Horário da criação do arquivo.

No Solaris e derivados, os seguintes atributos também podem estar disponíveis:

st_fstype
String que identifica exclusivamente o tipo de sistema de arquivos que contém o arquivo.

Em sistemas Mac OS, os seguintes atributos também podem estar disponíveis:

st_rsize
Tamanho real do arquivo.

st_creator
Criador do arquivo.

st_type
Tipo de arquivo.

On Windows systems, the following attribute is also available:

st_file_attributes
Atributos de arquivos no Windows: membro `dwFileAttributes` da estrutura `BY_HANDLE_FILE_INFORMATION` retornada por `GetFileInformationByHandle()`. Veja as constantes `FILE_ATTRIBUTE_*` no módulo *stat*.

O módulo padrão *stat* define funções e constantes que são úteis para extrair informações de uma estrutura `stat`. (No Windows, alguns itens são preenchidos com valores fictícios.)

Para compatibilidade com versões anteriores, uma instância de *stat_result* também é acessível como uma tupla de pelo menos 10 inteiros, fornecendo os membros mais importantes (e portáveis) da estrutura `stat`, na ordem *st_mode*, *st_ino*, *st_dev*, *st_nlink*, *st_uid*, *st_gid*, *st_size*, *st_atime*, *st_mtime*, *st_ctime*. Mais itens podem ser adicionados no final por algumas implementações. Para compatibilidade com versões mais antigas do Python, acessar *stat_result* como uma tupla sempre retorna inteiros.

Novo na versão 3.3: Adicionados os membros *st_atime_ns*, *st_mtime_ns* e *st_ctime_ns*.

Novo na versão 3.5: Adicionado o membro *st_file_attributes* no Windows.

Alterado na versão 3.5: Windows agora retorna o arquivo de índice como *st_ino* quando disponível.

Novo na versão 3.7: Adicionado o membro *st_fstype* ao Solaris/derivados.

`os.`**statvfs**(*path*)
Executa uma chamada de sistema `statvfs()` no caminho fornecido. O valor de retorno é um objeto cujos atributos descrevem o sistema de arquivos no caminho fornecido e correspondem aos membros da estrutura `statvfs`, a saber: `f_bsize`, `f_frsize`, `f_blocks`, `f_bfree`, `f_bavail`, `f_files`, `f_ffree`, `f_favail`, `f_flag`, `f_namemax`, `f_fsid`.

Duas constantes de nível de módulo são definidas para os sinalizadores de bit do atributo `f_flag`: se `ST_RDONLY` for definido, o sistema de arquivos é montado somente leitura, e se `ST_NOSUID` estiver definido, a semântica dos bits setuid/setgid é desabilitada ou não é suportada.

Constantes de nível de módulo adicionais são definidas para sistemas baseados em GNU/glibc. São elas `ST_NODEV` (impede o acesso aos arquivos especiais do dispositivo), `ST_NOEXEC` (não permite a execução do programa), `ST_SYNCHRONOUS` (as escritas são sincronizadas de uma vez), `ST_MANDLOCK` (permitir bloqueios obrigatórios em um sistema de arquivos), `ST_WRITE` (escrever no arquivo/diretório/symlink), `ST_APPEND` (arquivo somente anexado), `ST_IMMUTABLE` (arquivo imutável), `ST_NOATIME` (não atualiza os tempos de acesso), `ST_NODIRATIME` (não atualiza os tempos de acesso ao diretório), `ST_RELATIME` (atualiza o atime relativo ao mtime/ctime).

Esta função tem suporte a *especificar um descritor de arquivo*.

Disponibilidade: Unix.

Alterado na versão 3.2: As constantes `ST_RDONLY` e `ST_NOSUID` foram adicionadas.

Novo na versão 3.3: Added support for specifying an open file descriptor for *path*.

Alterado na versão 3.4: As constantes `ST_NODEV`, `ST_NOEXEC`, `ST_SYNCHRONOUS`, `ST_MANDLOCK`, `ST_WRITE`, `ST_APPEND`, `ST_IMMUTABLE`, `ST_NOATIME`, `ST_NODIRATIME` e `ST_RELATIME` foram adicionadas.

Alterado na versão 3.6: Aceita um *path-like object*.

Novo na versão 3.7: Adicionado `f_fsid`.

os.**supports_dir_fd**
A *Set* object indicating which functions in the *os* module permit use of their *dir_fd* parameter. Different platforms provide different functionality, and an option that might work on one might be unsupported on another. For consistency's sakes, functions that support *dir_fd* always allow specifying the parameter, but will raise an exception if the functionality is not actually available.

To check whether a particular function permits use of its *dir_fd* parameter, use the `in` operator on `supports_dir_fd`. As an example, this expression determines whether the *dir_fd* parameter of *os.stat()* is locally available:

```
os.stat in os.supports_dir_fd
```

Atualmente os parâmetros *dir_fd* funcionam apenas em plataformas Unix; nenhum deles funciona no Windows.

Novo na versão 3.3.

os.**supports_effective_ids**
A *Set* object indicating which functions in the *os* module permit use of the *effective_ids* parameter for *os.access()*. If the local platform supports it, the collection will contain *os.access()*, otherwise it will be empty.

To check whether you can use the *effective_ids* parameter for *os.access()*, use the `in` operator on `supports_effective_ids`, like so:

```
os.access in os.supports_effective_ids
```

Currently *effective_ids* only works on Unix platforms; it does not work on Windows.

Novo na versão 3.3.

os.**supports_fd**
A *Set* object indicating which functions in the *os* module permit specifying their *path* parameter as an open file descriptor. Different platforms provide different functionality, and an option that might work on one might be unsupported on another. For consistency's sakes, functions that support *fd* always allow specifying the parameter, but will raise an exception if the functionality is not actually available.

To check whether a particular function permits specifying an open file descriptor for its *path* parameter, use the `in` operator on `supports_fd`. As an example, this expression determines whether *os.chdir()* accepts open file descriptors when called on your local platform:

```
os.chdir in os.supports_fd
```

Novo na versão 3.3.

os.**supports_follow_symlinks**
A *Set* object indicating which functions in the *os* module permit use of their *follow_symlinks* parameter. Different platforms provide different functionality, and an option that might work on one might be unsupported on another. For consistency's sakes, functions that support *follow_symlinks* always allow specifying the parameter, but will raise an exception if the functionality is not actually available.

To check whether a particular function permits use of its *follow_symlinks* parameter, use the `in` operator on `supports_follow_symlinks`. As an example, this expression determines whether the *follow_symlinks* parameter of *os.stat()* is locally available:

```
os.stat in os.supports_follow_symlinks
```

Novo na versão 3.3.

os.**symlink**(*src*, *dst*, *target_is_directory=False*, ***, *dir_fd=None*)
Cria um link simbólico apontando para *src* chamado *dst*.

No Windows, um link simbólico representa um arquivo ou um diretório e não se transforma no destino dinamicamente. Se o alvo estiver presente, o tipo de link simbólico será criado para corresponder. Caso contrário, o link simbólico será criado como um diretório se *target_is_directory* for `True` ou um link simbólico de arquivo (o padrão) caso contrário. Em plataformas não Windows, *target_is_directory* é ignorado.

Symbolic link support was introduced in Windows 6.0 (Vista). *symlink()* will raise a *NotImplementedError* on Windows versions earlier than 6.0.

Esta função pode suportar *paths relative to directory descriptors*.

Nota: On Windows, the *SeCreateSymbolicLinkPrivilege* is required in order to successfully create symlinks. This privilege is not typically granted to regular users but is available to accounts which can escalate privileges to the administrator level. Either obtaining the privilege or running your application as an administrator are ways to successfully create symlinks.

A exceção *OSError* é levantada quando a função é chamada por um usuário sem privilégios.

Disponibilidade: Unix, Windows.

Alterado na versão 3.2: Adicionado suporte para links simbólicos do Windows 6.0 (Vista).

Novo na versão 3.3: Adicionado o argumento *dir_fd* e agora permite *target_is_directory* em plataformas não Windows.

Alterado na versão 3.6: Aceita o termos a *path-like object* para *src* e *dst*.

`os.`**`sync`**`()`
Força a escrita de tudo para o disco.

Disponibilidade: Unix.

Novo na versão 3.3.

`os.`**`truncate`**`(`*path*, *length*`)`
Trunca o arquivo correspondente ao *path*, de modo que tenha no máximo *length* bytes.

Esta função tem suporte a *especificar um descritor de arquivo*.

Disponibilidade: Unix, Windows.

Novo na versão 3.3.

Alterado na versão 3.5: Adicionado suporte para o Windows.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.`**`unlink`**`(`*path*, *, *dir_fd=None*`)`
Remove (exclui) o arquivo *path*. Esta função é semanticamente idêntica à *remove()*; o nome `unlink` é seu nome Unix tradicional. Por favor, veja a documentação de *remove()* para mais informações.

Novo na versão 3.3: O parâmetro *dir_fd*.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.`**`utime`**`(`*path*, *times=None*, **[*, *ns*`]`, *dir_fd=None*, *follow_symlinks=True*`)`
Define os tempos de acesso e modificação do arquivo especificado por *path*.

utime() aceita dois parâmetros opcionais, *times* e *ns*. Eles especificam os horários definidos no *path* e são usados da seguinte forma:

- Se *ns* for especificado, deve ser uma tupla de 2 elementos na forma `(atime_ns, mtime_ns)` onde cada membro é um int expressando nanossegundos.

- Se *times* não for `None`, deve ser uma tupla de 2 elementos na forma `(atime, mtime)` onde cada membro é um int ou ponto flutuante expressando segundos.
- Se *times* for `None` e *ns* não for especificado, isso é equivalente a especificar `ns=(atime_ns, mtime_ns)` onde ambos os tempos são a hora atual.

É um erro especificar tuplas para ambos *times* e *ns*.

Whether a directory can be given for *path* depends on whether the operating system implements directories as files (for example, Windows does not). Note that the exact times you set here may not be returned by a subsequent *stat()* call, depending on the resolution with which your operating system records access and modification times; see *stat()*. The best way to preserve exact times is to use the *st_atime_ns* and *st_mtime_ns* fields from the *os.stat()* result object with the *ns* parameter to *utime*.

Esta função pode ter suporte a *especificar um descritor de arquivo*, *caminhos relativos aos descritores de diretório* e *não seguir os links simbólicos*.

Novo na versão 3.3: Added support for specifying an open file descriptor for *path*, and the *dir_fd*, *follow_symlinks*, and *ns* parameters.

Alterado na versão 3.6: Aceita um *path-like object*.

`os.`**`walk`**(*top*, *topdown=True*, *onerror=None*, *followlinks=False*)

Gera os nomes dos arquivos em uma árvore de diretório percorrendo a árvore de cima para baixo ou de baixo para cima. Para cada diretório na árvore com raiz no diretório *top* (incluindo o próprio *top*), ele produz uma tupla de 3 elementos `(dirpath, dirnames, filenames)`.

dirpath is a string, the path to the directory. *dirnames* is a list of the names of the subdirectories in *dirpath* (excluding `'.'` and `'..'`). *filenames* is a list of the names of the non-directory files in *dirpath*. Note that the names in the lists contain no path components. To get a full path (which begins with *top*) to a file or directory in *dirpath*, do `os.path.join(dirpath, name)`.

Se o argumento opcional *topdown* for `True` ou não especificado, o triplo para um diretório é gerado antes dos triplos para qualquer um de seus subdiretórios (os diretórios são gerados de cima para baixo). Se *topdown* for `False`, o triplo para um diretório é gerado após os triplos para todos os seus subdiretórios (os diretórios são gerados de baixo para cima). Não importa o valor de *topdown*, a lista de subdiretórios é recuperada antes que as tuplas para o diretório e seus subdiretórios sejam geradas.

Quando *topdown* é `True`, o chamador pode modificar a lista de *dirnames* no local (talvez usando `del` ou atribuição de fatia), e *walk()* só recursará nos subdiretórios cujos nomes permanecem em *dirnames*; isso pode ser usado para podar a busca, impor uma ordem específica de visita, ou mesmo informar à função *walk()* sobre os diretórios que o chamador cria ou renomeia antes de retomar *walk()* novamente. Modificar *dirnames* quando *topdown* for `False` não tem efeito no comportamento da caminhada, porque no modo de baixo para cima os diretórios em *dirnames* são gerados antes do próprio *dirpath* ser gerado.

Por padrão, os erros da chamada de *scandir()* são ignorados. Se o argumento opcional *onerror* for especificado, deve ser uma função; ela será chamado com um argumento, uma instância da exceção *OSError*. Ele pode relatar o erro para continuar com a caminhada ou levantar a exceção para abortar a caminhada. Observe que o nome do arquivo está disponível como o atributo `filename` do objeto de exceção.

Por padrão, a função *walk()* não vai seguir links simbólicos que resolvem para diretórios. Defina *followlinks* como `True` para visitar diretórios apontados por links simbólicos, em sistemas que oferecem suporte a eles.

Nota: Esteja ciente de que definir *followlinks* para `True` pode levar a recursão infinita se um link apontar para um diretório pai de si mesmo. *walk()* não mantém registro dos diretórios que já visitou.

Nota: Se você passar um nome de caminho relativo, não mude o diretório de trabalho atual entre as continuações

de *walk()*. : func: *walk* nunca muda o diretório atual, e presume que seu chamador também não.

Este exemplo exibe o número de bytes obtidos por arquivos não pertencentes ao diretório em cada diretório no diretório inicial, exceto que ele não olha em nenhum subdiretório CVS:

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files), end=" ")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

No próximo exemplo (implementação simples de *shutil.rmtree()*), andar na árvore de baixo para cima é essencial, *rmdir()* não permite excluir um diretório antes que o diretório esteja vazio:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

Alterado na versão 3.5: Esta função agora chama *os.scandir()* em vez de *os.listdir()*, tornando-a mais rápida reduzindo o número de chamadas a *os.stat()*.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**fwalk**(*top='.'*, *topdown=True*, *onerror=None*, ***, *follow_symlinks=False*, *dir_fd=None*)

Se comporta exatamente como *walk()*, exceto que produz um tupla de 4 elementos `(dirpath, dirnames, filenames, dirfd)`, e tem suporte a `dir_fd`.

dirpath, *dirnames* e *filenames* são idênticos à saída de *walk()* e *dirfd* é um descritor de arquivo que faz referência ao diretório *dirpath*.

Esta função sempre tem suporte a *caminhos relativos aos descritores de diretório* e *não seguir links simbólicos*. Observe, entretanto, que, ao contrário de outras funções, o valor padrão *fwalk()* para *follow_symlinks* é `False`.

Nota: Uma vez que *fwalk()* produz descritores de arquivo, eles só são válidos até a próxima etapa de iteração, então você deve duplicá-los (por exemplo, com *dup()*) se quiser mantê-los por mais tempo.

Este exemplo exibe o número de bytes obtidos por arquivos não pertencentes ao diretório em cada diretório no diretório inicial, exceto que ele não olha em nenhum subdiretório CVS:

```
import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/email'):
    print(root, "consumes", end="")
    print(sum([os.stat(name, dir_fd=rootfd).st_size for name in files]),
          end="")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

No próximo exemplo, percorrer a árvore de baixo para cima é essencial: *rmdir()* não permite excluir um diretório antes que o diretório esteja vazio:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files, rootfd in os.fwalk(top, topdown=False):
    for name in files:
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)
```

Disponibilidade: Unix.

Novo na versão 3.3.

Alterado na versão 3.6: Aceita um *path-like object*.

Alterado na versão 3.7: Adicionado suporte para caminhos em *bytes*.

Atributos estendidos do Linux

Novo na versão 3.3.

Estas funções estão todas disponíveis apenas no Linux.

os.**getxattr**(*path*, *attribute*, *, *follow_symlinks=True*)
Retorna o valor do atributo estendido do sistema de arquivos *attribute* para *path*. *attribute* pode ser bytes ou str (direta ou indiretamente por meio da interface *PathLike*). Se for str, ele é codificado com a codificação do sistema de arquivos.

Esta função tem suporte a *especificar um descritor de arquivo* e *não seguir links simbólicos*.

Alterado na versão 3.6: Aceita um *objeto caminho ou similar* para *path* e *attribute*.

os.**listxattr**(*path=None*, *, *follow_symlinks=True*)
Retorna uma lista dos atributos estendidos do sistema de arquivos em *path*. Os atributos na lista são representados como strings decodificadas com a codificação do sistema de arquivos. Se *path* for None, *listxattr()* irá examinar o diretório atual.

Esta função tem suporte a *especificar um descritor de arquivo* e *não seguir links simbólicos*.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**removexattr**(*path*, *attribute*, *, *follow_symlinks=True*)
Remove o atributo estendido do sistema de arquivos *attribute* do *path*. *attribute* deve ser bytes ou str (direta ou indiretamente por meio da interface *PathLike*). Se for uma string, ele é codificado com a codificação do sistema de arquivos.

Esta função tem suporte a *especificar um descritor de arquivo* e *não seguir links simbólicos*.

Alterado na versão 3.6: Aceita um *objeto caminho ou similar* para *path* e *attribute*.

os.**setxattr**(*path*, *attribute*, *value*, *flags=0*, *, *follow_symlinks=True*)
Define o atributo de sistema de arquivos estendido *attribute* em *path* como *value*. *attribute* deve ser um bytes ou str sem NULs embutidos (direta ou indiretamente por meio da interface *PathLike*). Se for um str, ele é codificado com a codificação do sistema de arquivos. *sinalizadores* podem ser *XATTR_REPLACE* ou *XATTR_CREATE*. Se *XATTR_REPLACE* é fornecido e o atributo não existe, EEXISTS será levantado. Se *XATTR_CREATE* for fornecido e o atributo já existir, o atributo não será criado e ENODATA será levantado.

Esta função tem suporte a *especificar um descritor de arquivo* e *não seguir links simbólicos*.

Nota: Um bug nas versões do kernel Linux inferiores a 2.6.39 fez com que o argumento *flags* fosse ignorado em alguns sistemas de arquivos.

Alterado na versão 3.6: Aceita um *objeto caminho ou similar* para *path* e *attribute*.

os.**XATTR_SIZE_MAX**
O tamanho máximo que o valor de um atributo estendido pode ter. Atualmente, são 64 KiB no Linux.

os.**XATTR_CREATE**
Este é um valor possível para o argumento *flags* em `setxattr()`. Indica que a operação deve criar um atributo.

os.**XATTR_REPLACE**
Este é um valor possível para o argumento *flags* em `setxattr()`. Indica que a operação deve substituir um atributo existente.

16.1.6 Gerenciamento de processo

Estas funções podem ser usadas para criar e gerenciar processos.

As várias funções `exec*` pegam uma lista de argumentos para o novo programa carregado no processo. Em cada caso, o primeiro desses argumentos é passado para o novo programa como seu próprio nome, e não como um argumento que um usuário pode ter digitado em uma linha de comando. Para o programador C, este é o `argv[0]` passado para um programa `main()`. Por exemplo, `os.execv('/bin/echo', ['foo', 'bar'])` imprimirá apenas `bar` na saída padrão; `foo` parecerá ser ignorado.

os.**abort**()
Gera um sinal `SIGABRT` para o processo atual. No Unix, o comportamento padrão é produzir um despejo de memória; no Windows, o processo retorna imediatamente um código de saída `3`. Esteja ciente de que chamar esta função não chamará o manipulador de sinal Python registrado para `SIGABRT` com `signal.signal()`.

os.**execl**(*path*, *arg0*, *arg1*, ...)
os.**execle**(*path*, *arg0*, *arg1*, ..., *env*)
os.**execlp**(*file*, *arg0*, *arg1*, ...)
os.**execlpe**(*file*, *arg0*, *arg1*, ..., *env*)
os.**execv**(*path*, *args*)
os.**execve**(*path*, *args*, *env*)
os.**execvp**(*file*, *args*)
os.**execvpe**(*file*, *args*, *env*)
Todas essas funções executam um novo programa, substituindo o processo atual; eles não voltam. No Unix, o novo executável é carregado no processo atual e terá a mesma identificação de processo do chamador. Os erros serão relatados como exceções de `OSError`.

O processo atual é substituído imediatamente. Objetos de arquivos abertos e descritores não são descarregados, então se houver dados em buffer nesses arquivos abertos, você deve descarregá-los usando `sys.stdout.flush()` ou `os.fsync()` antes de chamar uma função `exec*`.

As variantes "l" e "v" das funções `exec*` diferem em como os argumentos da linha de comando são passados. As variantes "l" são talvez as mais fáceis de trabalhar se o número de parâmetros for fixo quando o código for escrito; os parâmetros individuais simplesmente se tornam parâmetros adicionais para as funções `execl*()`. As variantes "v" são boas quando o número de parâmetros é variável, com os argumentos sendo passados em uma lista ou tupla como o parâmetro *args*. Em qualquer caso, os argumentos para o processo filho devem começar com o nome do comando que está sendo executado, mas isso não é obrigatório.

As variantes que incluem um "p" próximo ao final (*execlp()*, *execlpe()*, *execvp()* e *execvpe()*) usarão a variável de ambiente PATH para localizar o programa *file*. Quando o ambiente está sendo substituído (usando uma das variantes *exec*e*, discutidas no próximo parágrafo), o novo ambiente é usado como fonte da variável PATH As outras variantes, *execl()*, *execle()*, *execv()* e *execve()*, não usarão a variável PATH para localizar o executável; *path* deve conter um caminho absoluto ou relativo apropriado.

Para *execle()*, *execlpe()*, *execve()* e *execvpe()* (observe que todos eles terminam em "e"), o parâmetro *env* deve ser um mapeamento que é usado para definir as variáveis de ambiente para o novo processo (elas são usadas no lugar do ambiente do processo atual); as funções *execl()*, *execlp()*, *execv()* e *execvp()* fazem com que o novo processo herde o ambiente do processo atual.

Para *execve()* em algumas plataformas, *path* também pode ser especificado como um descritor de arquivo aberto. Esta funcionalidade pode não ser compatível com sua plataforma; você pode verificar se está ou não disponível usando *os.supports_fd*. Se não estiver disponível, usá-lo vai levantar uma *NotImplementedError*.

Disponibilidade: Unix, Windows.

Novo na versão 3.3: Added support for specifying an open file descriptor for *path* for *execve()*.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**_exit**(*n*)
Sai do processo com status *n*, sem chamar manipuladores de limpeza, liberando buffers de stdio etc.

Nota: A forma padrão de sair é sys.exit(n). *_exit()* normalmente só deve ser usado no processo filho após uma função *fork()*.

Os seguintes códigos de saída são definidos e podem ser usados com *_exit()*, embora não sejam obrigatórios. Eles são normalmente usados para programas de sistema escritos em Python, como um programa de entrega de comando externo de servidor de e-mail.

Nota: Alguns deles podem não estar disponíveis em todas as plataformas Unix, pois há algumas variações. Essas constantes são definidas onde são definidas pela plataforma subjacente.

---

os.**EX_OK**
Código de saída que significa que ocorreu nenhum erro.

Disponibilidade: Unix.

os.**EX_USAGE**
Código de saída que significa que o comando foi usado incorretamente, como quando o número errado de argumentos é fornecido.

Disponibilidade: Unix.

os.**EX_DATAERR**
Código de saída que significa que os dados inseridos estavam incorretos.

Disponibilidade: Unix.

os.**EX_NOINPUT**
Código de saída que significa que um arquivo de entrada não existe ou não pôde ser lido.

Disponibilidade: Unix.

os.**EX_NOUSER**
Código de saída que significa que um usuário especificado não existe.

Disponibilidade: Unix.

os.**EX_NOHOST**
Código de saída que significa que um host especificado não existe.

Disponibilidade: Unix.

os.**EX_UNAVAILABLE**
Código de saída que significa que um serviço necessário está indisponível.

Disponibilidade: Unix.

os.**EX_SOFTWARE**
Código de saída que significa que um erro interno do software foi detectado.

Disponibilidade: Unix.

os.**EX_OSERR**
Código de saída que significa que um erro do sistema operacional foi detectado, como a incapacidade de fazer fork ou criar um encadeamento.

Disponibilidade: Unix.

os.**EX_OSFILE**
Código de saída que significa que algum arquivo do sistema não existia, não pôde ser aberto ou teve algum outro tipo de erro.

Disponibilidade: Unix.

os.**EX_CANTCREAT**
Código de saída que significa que um arquivo de saída especificado pelo usuário não pôde ser criado.

Disponibilidade: Unix.

os.**EX_IOERR**
Código de saída que significa que ocorreu um erro ao fazer E/S em algum arquivo.

Disponibilidade: Unix.

os.**EX_TEMPFAIL**
Código de saída que significa que ocorreu uma falha temporária. Isso indica algo que pode não ser realmente um erro, como uma conexão de rede que não pôde ser feita durante uma operação de nova tentativa.

Disponibilidade: Unix.

os.**EX_PROTOCOL**
Código de saída que significa que uma troca de protocolo foi ilegal, inválida ou não compreendida.

Disponibilidade: Unix.

os.**EX_NOPERM**
Código de saída que significa que não havia permissões suficientes para executar a operação (mas sem intenção de causar problemas do sistema de arquivos).

Disponibilidade: Unix.

os.**EX_CONFIG**
Código de saída que significa que ocorreu algum tipo de erro de configuração.

Disponibilidade: Unix.

os.**EX_NOTFOUND**
Código de saída que significa algo como “uma entrada não foi encontrada”.

Disponibilidade: Unix.

os.**fork**()
Faz um fork de um processo filho. Retorna `0` no filho e o ID de processo do filho no pai. Se ocorrer um erro, uma *OSError* é levantada.

Note that some platforms including FreeBSD <= 6.3 and Cygwin have known issues when using fork() from a thread.

> **Aviso:** Veja *ssl* para aplicações que usam o módulo SSL com fork().

Disponibilidade: Unix.

os.**forkpty**()
Faz um fork de um processo filho, usando um novo pseudoterminal como o terminal de controle do filho. Retorna um par de `(pid, fd)`, onde *pid* é `0` no filho, o novo ID de processo do filho no pai e *fd* é o descritor de arquivo do lado mestre do pseudoterminal. Para uma abordagem mais portátil, use o módulo *pty*. Se ocorrer um erro, *OSError* é levantada.

Availability: alguns tipos de Unix.

os.**kill**(*pid*, *sig*)
Envia o sinal *sig* para o processo *pid*. Constantes dos sinais específicos disponíveis na plataforma host são definidas no módulo *signal*.

Windows: Os sinais *signal.CTRL_C_EVENT* e *signal.CTRL_BREAK_EVENT* são sinais especiais que só podem ser enviados para processos de console que compartilham uma janela de console comum, por exemplo, alguns subprocessos. Qualquer outro valor para *sig* fará com que o processo seja encerrado incondicionalmente pela API TerminateProcess e o código de saída será definido como *sig*. A versão Windows de *kill()* adicionalmente leva identificadores de processo para ser morto.

Veja também *signal.pthread_kill()*.

Novo na versão 3.2: Suporte ao Windows.

os.**killpg**(*pgid*, *sig*)
Envia o sinal *sig* para o grupo de processo *pgid*.

Disponibilidade: Unix.

os.**nice**(*increment*)
Adiciona *increment* ao nível de “nice” do processo. Retorna um novo nível de “nice”.

Disponibilidade: Unix.

os.**plock**(*op*)
Bloqueia os segmentos do programa na memória. O valor de *op* (definido em `<sys/lock.h>`) determina quais segmentos estão bloqueados.

Disponibilidade: Unix.

os.**popen**(*cmd*, *mode='r'*, *buffering=-1*)
Abre um encadeamento, também conhecido como “pipe”, de ou para o comando *cmd*. O valor de retorno é um objeto arquivo aberto conectado ao encadeamento, que pode ser lido ou escrito dependendo se *mode* é `'r'` (padrão) ou `'w'`. O argumento *buffering* tem o mesmo significado que o argumento correspondente para a função embutida *open()*. O objeto arquivo retornado lê ou escreve strings de texto em vez de bytes.

O método `close` retorna *None* se o subprocesso foi encerrado com sucesso, ou o código de retorno do subprocesso se houve um erro. Em sistemas POSIX, se o código de retorno for positivo, ele representa o valor de retorno do processo deslocado para a esquerda em um byte. Se o código de retorno for negativo, o processo foi encerrado

pelo sinal dado pelo valor negado do código de retorno. (Por exemplo, o valor de retorno pode ser `- signal.SIGKILL` se o subprocesso foi eliminado.) Em sistemas Windows, o valor de retorno contém o código de retorno inteiro com sinal do processo filho.

Isso é implementado usando *subprocess.Popen*; consulte a documentação desta classe para maneiras mais poderosas de gerenciar e se comunicar com subprocessos.

os.**register_at_fork**(*, *before=None*, *after_in_parent=None*, *after_in_child=None*)

Registra chamáveis a serem executados quando um novo processo filho é criado usando *os.fork()* ou APIs de clonagem de processos semelhantes. Os parâmetros são opcionais e somente-nomeados. Cada um especifica um ponto de chamada diferente.

- *before* é uma função chamada antes de fazer um fork para um processo filho.
- *after_in_parent* é uma função chamada a partir do processo pai após fazer um fork para um processo filho.
- *after_in_child* é uma função chamada a partir do processo filho.

Essas chamadas são feitas apenas se o controle deve retornar ao interpretador Python. Um lançamento típico de *subprocess* não irá acioná-los, pois o filho não entrará novamente no interpretador.

As funções registradas para execução antes de fazer o fork são chamadas na ordem de registro reversa. As funções registradas para execução após o fork ser feito (no pai ou no filho) são chamadas na ordem de registro.

Note que chamadas a `fork()` feitas por código C de terceiros pode não chamar estas funções, a menos que ele explicitamente chame `PyOS_BeforeFork()`, `PyOS_AfterFork_Parent()` e `PyOS_AfterFork_Child()`.

Não há uma forma de desfazer o registro de uma função.

Disponibilidade: Unix.

Novo na versão 3.7.

os.**spawnl**(*mode*, *path*, *...*)
os.**spawnle**(*mode*, *path*, *...*, *env*)
os.**spawnlp**(*mode*, *file*, *...*)
os.**spawnlpe**(*mode*, *file*, *...*, *env*)
os.**spawnv**(*mode*, *path*, *args*)
os.**spawnve**(*mode*, *path*, *args*, *env*)
os.**spawnvp**(*mode*, *file*, *args*)
os.**spawnvpe**(*mode*, *file*, *args*, *env*)

Executa o programa *path* em um novo processo.

(Observe que o módulo *subprocess* fornece recursos mais poderosos para gerar novos processos e recuperar seus resultados; usar esse módulo é preferível a usar essas funções. Verifique especialmente a seção *Replacing Older Functions with the subprocess Module*.)

Se *mode* for *P_NOWAIT*, esta função retorna o id do processo do novo processo; se *mode* for *P_WAIT*, retorna o código de saída do processo se ele sair normalmente, ou `-signal`, onde *signal* é o sinal que matou o processo. No Windows, o id de processo será na verdade o identificador do processo, portanto, pode ser usado com a função *waitpid()*.

As variantes "l" e "v" das funções *spawn** diferem em como os argumentos de linha de comando são passados. As variantes "l" são talvez as mais fáceis de trabalhar se o número de parâmetros for fixo quando o código for escrito; os parâmetros individuais simplesmente se tornam parâmetros adicionais para as funções `spawnl*()`. As variantes "v" são boas quando o número de parâmetros é variável, com os argumentos sendo passados em uma lista ou tupla como o parâmetro *args*. Em ambos os casos, os argumentos para o processo filho devem começar com o nome do comando que está sendo executado.

As variantes que incluem um segundo "p" próximo ao final (*spawnlp()*, *spawnlpe()*, *spawnvp()* e *spawnvpe()*) usarão a variável de ambiente `PATH` para localizar o programa *file*. Quando o ambiente está

sendo substituído (usando uma das variantes *spawn*e*, discutidas no próximo parágrafo), o novo ambiente é usado como fonte da variável PATH As outras variantes, *spawnl()*, *spawnle()*, *spawnv()* e *spawnve()*, não usarão a variável PATH para localizar o executável; *path* deve conter um caminho absoluto ou relativo apropriado.

Para *spawnle()*, *spawnlpe()*, *spawnve()* e *spawnvpe()* (observe que todos eles terminam em "e"), o parâmetro *env* deve ser um mapeamento que é usado para definir as variáveis de ambiente para o novo processo (elas são usadas no lugar do ambiente do processo atual); as funções *spawnl()*, *spawnlp()*, *spawnv()* e *spawnvp()* fazem com que o novo processo herde o ambiente do processo atual. Note que as chaves e os valores no dicionário *env* devem ser strings; chaves ou valores inválidos farão com que a função falhe, com um valor de retorno de 127.

Como exemplo, as seguintes chamadas a *spawnlp()* e *spawnvpe()* são equivalentes:

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

Disponibilidade: Unix, Windows. *spawnlp()*, *spawnlpe()*, *spawnvp()* e *spawnvpe()* não estão disponíveis no Windows. *spawnle()* e *spawnve()* não são seguros para thread no Windows; recomendamos que você use o módulo *subprocess*.

Alterado na versão 3.6: Aceita um *path-like object*.

os.**P_NOWAIT**
os.**P_NOWAITO**
Valores possíveis para o parâmetro *mode* para a família de funções *spawn**. Se qualquer um desses valores for fornecido, as funções spawn*() retornarão assim que o novo processo for criado, com o id do processo como o valor de retorno.

Disponibilidade: Unix, Windows.

os.**P_WAIT**
Valor possível para o parâmetro *mode* para a família de funções *spawn**. Se for fornecido como *mode*, as funções spawn*() não retornarão até que o novo processo seja executado até a conclusão e retornará o código de saída do processo em que a execução foi bem-sucedida, ou -signal se um sinal interromper o processo.

Disponibilidade: Unix, Windows.

os.**P_DETACH**
os.**P_OVERLAY**
Valores possíveis do o parâmetro *mode* para a família de funções *spawn**. Eles são menos portáteis do que os listados acima. *P_DETACH* é semelhante a *P_NOWAIT*, mas o novo processo é separado do console do processo de chamada. Se *P_OVERLAY* for usado, o processo atual será substituído; a função *spawn** não retornará.

Disponibilidade: Windows.

os.**startfile**(*path*[, *operation*])
Inicia um arquivo com sua aplicação associada.

Quando *operation* não é especificado ou não está 'open', isso atua como um clique duplo no arquivo no Windows Explorer, ou como fornecer o nome do arquivo como um argumento para o comando **start** do console interativo de comandos: o arquivo é aberto com qualquer aplicação (se houver) com a extensão associada.

Quando outra *operation* é fornecida, ela deve ser um "verbo de comando" que especifica o que deve ser feito com o arquivo. Verbos comuns documentados pela Microsoft são 'print' e 'edit' (para serem usados em arquivos), bem como 'explore' e 'find' (para serem usados em diretórios).

startfile() retorna assim que a aplicação associada é iniciada. Não há opção de aguardar o fechamento da aplicação e nenhuma maneira de recuperar o status de saída da aplicação. O parâmetro *path* é relativo ao diretório atual. Se você quiser usar um caminho absoluto, certifique-se de que o primeiro caractere não seja uma barra (`'/'`); a função Win32 `ShellExecute()` subjacente não funciona se for. Use a função *os.path.normpath()* para garantir que o caminho esteja devidamente codificado para Win32.

Para reduzir a sobrecarga de inicialização do interpretador, a função Win32 `ShellExecute()` não é resolvida até que esta função seja chamada pela primeira vez. Se a função não puder ser resolvida, *NotImplementedError* será levantada.

Disponibilidade: Windows.

`os.`**`system`**`(`*command*`)`

Executa o comando (uma string) em um subshell. Isso é implementado chamando a função C padrão `system()`, e tem as mesmas limitações. Alterações em *sys.stdin* etc. não são refletidas no ambiente do comando executado. Se *command* gerar qualquer saída, ela será enviada ao fluxo de saída padrão do interpretador.

No Unix, o valor de retorno é o status de saída do processo codificado no formato especificado para *wait()*. Observe que POSIX não especifica o significado do valor de retorno da função C `system()`, então o valor de retorno da função Python é dependente do sistema.

No Windows, o valor de retorno é aquele retornado pelo shell do sistema após a execução de *command*. O shell é fornecido pela variável de ambiente Windows `COMSPEC`: normalmente é **`cmd.exe`**, que retorna o status de saída da execução do comando; em sistemas que usam um shell não nativo, consulte a documentação do shell.

O módulo *subprocess* fornece recursos mais poderosos para gerar novos processos e recuperar seus resultados; usar esse módulo é preferível a usar esta função. Veja a seção *Replacing Older Functions with the subprocess Module* na documentação do *subprocess* para algumas receitas úteis.

Disponibilidade: Unix, Windows.

`os.`**`times`**`()`

Retorna os tempos atuais de processo globais. O valor de retorno é um objeto com cinco atributos:

- `user` - tempo do usuário
- *system* - tempo do sistema
- `children_user` - tempo do usuário de todos os processo filhos
- `children_system` - tempo do sistema de todos os processo filhos
- `elapsed` - tempo real decorrido desde um ponto fixo no passado

Para compatibilidade com versões anteriores, esse objeto também se comporta como uma tupla de 5 elementos contendo `user`, *system*, `children_user`, `children_system` e `elapsed`` nessa ordem.

Consulte as páginas de manual *times (2)* e *times (3)* no Unix ou o GetProcessTimes MSDN no Windows. No Windows, apenas `user` e *system* são conhecidos; os outros atributos são zero.

Disponibilidade: Unix, Windows.

Alterado na versão 3.3: Retorna tupla alterada de uma tupla para um objeto tipo tupla com atributos nomeados.

`os.`**`wait`**`()`

Aguarda a conclusão de um processo filho e retorna uma tupla contendo seu pid e indicação de status de saída: um número de 16 bits, cujo byte baixo é o número do sinal que matou o processo e cujo byte alto é o status de saída (se o sinal número é zero); o bit alto do byte baixo é definido se um arquivo principal foi produzido.

Disponibilidade: Unix.

`os.`**`waitid`**`(`*idtype*, *id*, *options*`)`

Wait for the completion of one or more child processes. *idtype* can be *P_PID*, *P_PGID* or *P_ALL*. *id* specifies the pid to wait on. *options* is constructed from the ORing of one or more of *WEXITED*, *WSTOPPED* or *WCONTINUED*

and additionally may be ORed with *WNOHANG* or *WNOWAIT*. The return value is an object representing the data contained in the `siginfo_t` structure, namely: `si_pid`, `si_uid`, `si_signo`, `si_status`, `si_code` or `None` if *WNOHANG* is specified and there are no children in a waitable state.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**P_PID**
os.**P_PGID**
os.**P_ALL**
Estes são os valores possíveis para *idtype* em *waitid()*. Eles afetam como *id* é interpretado.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**WEXITED**
os.**WSTOPPED**
os.**WNOWAIT**
Sinalizadores que podem ser usados em *options* em *waitid()* que especificam por qual sinal filho esperar.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**CLD_EXITED**
os.**CLD_DUMPED**
os.**CLD_TRAPPED**
os.**CLD_CONTINUED**
Estes são os valores possíveis para `si_code` no resultado retornado por *waitid()*.

Disponibilidade: Unix.

Novo na versão 3.3.

os.**waitpid**(*pid*, *options*)
Os detalhes desta função diferem no Unix e no Windows.

No Unix: Aguarda a conclusão de um processo filho dado pelo id de processo *pid* e retorna uma tupla contendo seu id de processo e indicação de status de saída (codificado como para *wait()*). A semântica da chamada é afetada pelo valor do inteiro *options*, que deve ser `0` para operação normal.

Se *pid for maior que "0", :func:'waitpid' solicita informações de status para aquele processo específico. Se *pid* for `0`, a solicitação é para o status de qualquer filho no grupo de processos do processo atual. Se *pid* for `-1`, a solicitação pertence a qualquer filho do processo atual. Se *pid* for menor que `-1`, o status é solicitado para qualquer processo no grupo de processos `-pid` (o valor absoluto de *pid*).

Uma *OSError* é levantada com o valor de errno quando a chamada de sistema retorna -1.

No Windows: Aguarda a conclusão de um processo fornecido pelo identificador de processo *pid* e retorna uma tupla contendo *pid*, e seu status de saída deslocado 8 bits para a esquerda (o deslocamento torna o uso da função em várias plataformas mais fácil). Um *pid* menor ou igual a `0` não tem nenhum significado especial no Windows e levanta uma exceção. O valor de inteiros *options* não tem efeito. *pid* pode se referir a qualquer processo cujo id é conhecido, não necessariamente um processo filho. As funções *spawn** chamadas com *P_NOWAIT* retornam manipuladores de processo adequados.

Alterado na versão 3.5: Se a chamada de sistema é interrompida e o tratador de sinal não levanta uma exceção, a função agora tenta novamente a chamada de sistema em vez de levantar uma exceção *InterruptedError* (consulte **PEP 475** para entender a lógica).

os.**wait3**(*options*)
Semelhante a *waitpid()*, exceto que nenhum argumento de id de processo é fornecido e uma tupla de 3 elementos contendo a id do processo da criança, indicação de status de saída e informações de uso de recursos é retornada. Consulte *resource.getrusage()* para obter detalhes sobre as informações de uso de recursos. O argumento da opção é o mesmo fornecido para *waitpid()* e *wait4()*.

Disponibilidade: Unix.

os.**wait4**(*pid*, *options*)
Semelhante a *waitpid()*, exceto uma tupla de 3 elementos, contendo a id do processo da filho, indicação de status de saída e informações de uso de recursos é retornada. Consulte *resource.getrusage()* para obter detalhes sobre as informações de uso de recursos. Os argumentos para *wait4()* são os mesmos que aqueles fornecidos a *waitpid()*.

Disponibilidade: Unix.

os.**WNOHANG**
A opção para *waitpid()* retornar imediatamente se nenhum status de processo filho estiver disponível imediatamente. A função retorna `(0, 0)` neste caso.

Disponibilidade: Unix.

os.**WCONTINUED**
Esta opção faz com que os processos filhos sejam relatados se eles foram continuados a partir de uma parada de controle de tarefa desde que seu status foi relatado pela última vez.

Disponibilidade: alguns sistemas Unix.

os.**WUNTRACED**
Esta opção faz com que os processos filhos sejam relatados se eles tiverem sido interrompidos, mas seu estado atual não foi relatado desde que foram interrompidos.

Disponibilidade: Unix.

As funções a seguir recebem um código de status do processo conforme retornado por *system()*, *wait()* ou *waitpid()* como parâmetro. Eles podem ser usados para determinar a disposição de um processo.

os.**WCOREDUMP**(*status*)
Retorna `True` se um despejo de núcleo *(core dump)* foi gerado para o processo; caso contrário, retorna `False`.

Esta função deve ser empregada apenas se *WIFSIGNALED()* for verdadeira.

Disponibilidade: Unix.

os.**WIFCONTINUED**(*status*)
Retorna `True` se um filho interrompido foi retomado pela entrega de *SIGCONT* (se o processo foi continuado de uma parada de controle de trabalho); caso contrário, retorna `False`.

Veja a opção *WCONTINUED*.

Disponibilidade: Unix.

os.**WIFSTOPPED**(*status*)
Retorna `True` se o processo foi interrompido pela entrega de um sinal; caso contrário, retorna `False`.

WIFSTOPPED() só retorna `True` se a chamada de *waitpid()* foi feita usando a opção *WUNTRACED* ou quando o processo está sendo rastreado (veja *ptrace(2)*).

Disponibilidade: Unix.

os.**WIFSIGNALED**(*status*)
Retorna `True` se o processo foi encerrado por um sinal; caso contrário, retorna `False`.

Disponibilidade: Unix.

os.**WIFEXITED**(*status*)
Retorna `True` se o processo foi encerrado normalmente, isto é, chamando `exit()` ou `_exit()`, ou retornando de `main()`; caso contrário, retorna `False`.

Disponibilidade: Unix.

os.**WEXITSTATUS**(*status*)
Retorna o status de saída do processo.

Esta função deve ser empregada apenas se *WIFEXITED()* for verdadeira.

Disponibilidade: Unix.

os.**WSTOPSIG**(*status*)
Retorna o sinal que causou a interrupção do processo.

Esta função deve ser empregada apenas se *WIFSTOPPED()* for verdadeira.

Disponibilidade: Unix.

os.**WTERMSIG**(*status*)
Retorna o número do sinal que causou o encerramento do processo.

Esta função deve ser empregada apenas se *WIFSIGNALED()* for verdadeira.

Disponibilidade: Unix.

16.1.7 Interface do agendador

Essas funções controlam como o tempo de CPU de um processo é alocado pelo sistema operacional. Eles estão disponíveis apenas em algumas plataformas Unix. Para informações mais detalhadas, consulte suas páginas man do Unix.

Novo na versão 3.3.

As políticas de agendamento a seguir serão expostas se houver suporte pelo sistema operacional.

os.**SCHED_OTHER**
A política de agendamento padrão.

os.**SCHED_BATCH**
Política de agendamento para processos com uso intensivo de CPU que tenta preservar a interatividade no resto do computador.

os.**SCHED_IDLE**
Política de agendamento para tarefas em segundo plano de prioridade extremamente baixa.

os.**SCHED_SPORADIC**
Política de agendamento para programas de servidor esporádicos.

os.**SCHED_FIFO**
Uma política de agendamento Primeiro a Entrar, Primeiro a Sair (FIFO).

os.**SCHED_RR**
Uma política de agendamento round-robin.

os.**SCHED_RESET_ON_FORK**
Este sinalizador pode ser operado em OU com qualquer outra política de agendamento. Quando um processo com este sinalizador definido bifurca, a política de agendamento e a prioridade de seu filho são redefinidas para o padrão.

class os.**sched_param**(*sched_priority*)
Esta classe representa parâmetros de agendamento ajustáveis usados em *sched_setparam()*, *sched_setscheduler()* e *sched_getparam()*. É imutável.

Neste momento, há somente um único parâmetro possível:

sched_priority
A prioridade de agendamento para uma política de agendamento.

os.**sched_get_priority_min**(*policy*)
Obtém o valor mínimo de prioridade para *policy*. *policy* é uma das constantes de política de agendamento acima.

os.**sched_get_priority_max**(*policy*)
Obtém o valor máximo de prioridade para *policy*. *policy* é uma das constantes de política de agendamento acima.

os.**sched_setscheduler**(*pid*, *policy*, *param*)
Define a política de agendamento para o processo com PID *pid*. Um *pid* de 0 significa o processo de chamada. *policy* é uma das constantes de política de agendamento acima. *param* é uma instância de *sched_param*.

os.**sched_getscheduler**(*pid*)
Retorna a política de agendamento para o processo com PID *pid*. Um *pid* de 0 significa o processo de chamada. O resultado é uma das constantes de política de agendamento acima.

os.**sched_setparam**(*pid*, *param*)
Define os parâmetros de agendamento para o processo com PID *pid*. Um *pid* de 0 significa o processo de chamada. *param* é uma instância de *sched_param*.

os.**sched_getparam**(*pid*)
Retorna os parâmetros de agendamento como uma instância de *sched_param* para o processo com PID *pid*. Um *pid* de 0 significa o processo de chamada.

os.**sched_rr_get_interval**(*pid*)
Retorna o quantum round-robin em segundos para o processo com PID *pid*. Um *pid* de 0 significa o processo de chamada.

os.**sched_yield**()
Libera a CPU voluntariamente.

os.**sched_setaffinity**(*pid*, *mask*)
Restringe o processo com PID *pid* (ou o processo atual se zero) para um conjunto de CPUs. *mask* é um iterável de inteiros que representam o conjunto de CPUs às quais o processo deve ser restrito.

os.**sched_getaffinity**(*pid*)
Retorna o conjunto de CPUs ao qual o processo com PID *pid* (ou o processo atual se zero) está restrito.

16.1.8 Diversas Informações de Sistema

os.**confstr**(*name*)
Retorna valores de configuração do sistema com valor de string. *name* especifica o valor de configuração a ser recuperado; pode ser uma string que é o nome de um valor de sistema definido; esses nomes são especificados em vários padrões (POSIX, Unix 95, Unix 98 e outros). Algumas plataformas também definem nomes adicionais. Os nomes conhecidos pelo sistema operacional host são fornecidos como as chaves do dicionário `confstr_names`. Para variáveis de configuração não incluídas nesse mapeamento, passar um número inteiro para *name* também é aceito.

Se o valor de configuração especificado por *name* não for definido, retorna `None`.

Se *name* for uma string e não for conhecida, *ValueError* é levantada. Se um valor específico para *name* não for compatível com o sistema de host, mesmo que seja incluído em `confstr_names`, uma *OSError* é levantada com *errno.EINVAL* como número do erro.

Disponibilidade: Unix.

os.**confstr_names**
Nomes de mapeamento de dicionário aceitos por *confstr()* para os valores inteiros definidos para esses nomes pelo sistema operacional do host. Isso pode ser usado para determinar o conjunto de nomes conhecidos pelo sistema.

Disponibilidade: Unix.

os.**cpu_count**()
Retorna o número de CPUs do sistema. Retorna None se não determinado.

Este número não é equivalente ao número de CPUs que o processo atual pode usar. O número de CPUs utilizáveis pode ser obtido com len(os.sched_getaffinity(0))

Novo na versão 3.4.

os.**getloadavg**()
Retorna o número de processos na fila de execução do sistema em média nos últimos 1, 5 e 15 minutos ou levanta *OSError* se a média de carga não foi obtida.

Disponibilidade: Unix.

os.**sysconf**(*name*)
Retorna valores de configuração do sistema com valor inteiro. Se o valor de configuração especificado por *name* não estiver definido, -1 é retornado. Os comentários sobre o parâmetro *name* para *confstr()* se aplicam aqui também; o dicionário que fornece informações sobre os nomes conhecidos é fornecido por sysconf_names.

Disponibilidade: Unix.

os.**sysconf_names**
Nomes de mapeamento de dicionário aceitos por *sysconf()* para os valores inteiros definidos para esses nomes pelo sistema operacional do host. Isso pode ser usado para determinar o conjunto de nomes conhecidos pelo sistema.

Disponibilidade: Unix.

Os dados a seguir são usados para suportar operações de manipulação de path. Estão definidos e disponíveis para todas as plataformas.

Operações de nível mais alto em nomes de caminho são definidos no módulo *os.path*.

os.**curdir**
A string constante usada pelo sistema operacional para se referir ao diretório atual. Isso é '.' para Windows e POSIX. Também disponível via *os.path*.

os.**pardir**
A string constante usada pelo sistema operacional para se referir ao diretório pai. Isso é '..' para Windows e POSIX. Também disponível via *os.path*.

os.**sep**
O caractere usado pelo sistema operacional para separar os componentes do nome do caminho. Este é '/' para POSIX e '\\' para Windows. Observe que saber disso não é suficiente para ser capaz de analisar ou concatenar nomes de caminho – use *os.path.split()* e *os.path.join()* – mas ocasionalmente é útil. Também disponível via *os.path*.

os.**altsep**
Um caractere alternativo usado pelo sistema operacional para separar os componentes do nome de caminho, ou None se apenas um caractere separador existir. Isso é definido como '/' em sistemas Windows onde sep é uma contrabarra. Também disponível via *os.path*.

os.**extsep**
O caractere que separa o nome do arquivo base da extensão; por exemplo, o '.' em os.py. Também disponível via *os.path*.

os.**pathsep**
O caractere convencionalmente usado pelo sistema operacional para separar os componentes do caminho de pesquisa (como em PATH), como ':' para POSIX ou ';' para Windows. Também disponível via *os.path*.

os.**defpath**
O caminho de pesquisa padrão usado por *exec*p** e *spawn*p** se o ambiente não tiver uma chave 'PATH'. Também disponível via *os.path*.

os.**linesep**
A string usada para separar (ou melhor, encerrar) linhas na plataforma atual. Pode ser um único caractere, como '\n' para POSIX, ou múltiplos caracteres, por exemplo, '\r\n' para Windows. Não use *os.linesep* como terminador de linha ao escrever arquivos abertos em modo de texto (o padrão); use um único '\n' ao invés, em todas as plataformas.

os.**devnull**
O caminho do arquivo do dispositivo nulo. Por exemplo: '/dev/null' para POSIX, 'nul' para Windows. Também disponível via *os.path*.

os.**RTLD_LAZY**
os.**RTLD_NOW**
os.**RTLD_GLOBAL**
os.**RTLD_LOCAL**
os.**RTLD_NODELETE**
os.**RTLD_NOLOAD**
os.**RTLD_DEEPBIND**
Sinalizadores para uso com as funções *setdlopenflags()* e *getdlopenflags()*. Veja a página man do Unix *dlopen(3)* para saber o que significam os diferentes sinalizadores.

Novo na versão 3.3.

16.1.9 Números aleatórios

os.**getrandom**(*size*, *flags=0*)
Obtém até *size* bytes aleatórios. Esta função pode retornar menos bytes que a quantia requisitada.

Esses bytes podem ser usados para propagar geradores de número aleatório no espaço do usuário ou para fins criptográficos.

getrandom() depende da entropia obtida de drivers de dispositivos e outras fontes de ruído ambiental. A leitura desnecessária de grandes quantidades de dados terá um impacto negativo sobre outros usuários dos dispositivos /dev/random e /dev/urandom.

O argumento sinalizadores é uma máscara de bits que pode conter zero ou mais dos seguintes valores operados com OU juntos: *os.GRND_RANDOM* e *GRND_NONBLOCK*.

Veja também A página do manual do Linux sobre getrandom().

Disponibilidade: Linux 3.17 e mais novos.

Novo na versão 3.6.

os.**urandom**(*size*)
Retorna uma string de *size* bytes aleatórios próprios para uso criptográfico.

Esta função retorna bytes aleatórios de uma fonte de aleatoriedade específica do sistema operacional. Os dados retornados devem ser imprevisíveis o suficiente para aplicações criptográficos, embora sua qualidade exata dependa da implementação do sistema operacional.

No Linux, se a chamada de sistema getrandom() estiver disponível, ela é usada no modo bloqueante: bloqueia até que o pool de entropia urandom do sistema seja inicializado (128 bits de entropia são coletados pelo kernel).

Veja a **PEP 524** para a justificativa. No Linux, a função *getrandom()* pode ser usada para obter bytes aleatórios no modo não bloqueante (usando a sinalização *GRND_NONBLOCK*) ou para pesquisar até que o pool de entropia urandom do sistema seja inicializado.

Em um sistema semelhante ao Unix, bytes aleatórios são lidos do dispositivo `/dev/urandom`. Se o dispositivo `/dev/urandom` não estiver disponível ou não for legível, a exceção *NotImplementedError* é levantada.

No Windows, será usado `CryptGenRandom()`.

Ver também:

O módulo *secrets* fornece funções de nível mais alto. Para uma interface fácil de usar para o gerador de números aleatórios fornecido por sua plataforma, consulte *random.SystemRandom*.

Alterado na versão 3.6.0: No Linux, `getrandom()` é usado agora no modo de bloqueio para aumentar a segurança.

Alterado na versão 3.5.2: No Linux, se a chamada de sistema `getrandom()` bloqueia (o pool de entropia urandom ainda não foi inicializado), recorre à leitura `/dev/urandom`.

Alterado na versão 3.5: No Linux 3.17 e mais recente, a chamada de sistema `getrandom()` agora é usada quando disponível. No OpenBSD 5.6 e mais recentes, a função C `getentropy()` agora é usada. Essas funções evitam o uso de um descritor de arquivo interno.

`os.`**`GRND_NONBLOCK`**

Por padrão, ao ler de `/dev/random`, *getrandom()* bloqueia se nenhum byte aleatório estiver disponível, e ao ler de `/dev/urandom`, ele bloqueia se o pool de entropia não ainda foi inicializado.

Se o sinalizador *GRND_NONBLOCK* estiver definido, então *getrandom()* não bloqueia nesses casos, mas, em vez disso, levanta *BlockingIOError* imediatamente.

Novo na versão 3.6.

`os.`**`GRND_RANDOM`**

Se este bit é definido bytes aleatórios são pegos a partir de `/dev/random` ao invés de `/dev/urandom`.

Novo na versão 3.6.

16.2 `io` — Ferramentas principais para trabalhar com fluxos

Código Fonte: Lib/io.py

16.2.1 Visão Geral

The *io* module provides Python's main facilities for dealing with various types of I/O. There are three main types of I/O: *text I/O*, *binary I/O* and *raw I/O*. These are generic categories, and various backing stores can be used for each of them. A concrete object belonging to any of these categories is called a *file object*. Other common terms are *stream* and *file-like object*.

Independent of its category, each concrete stream object will also have various capabilities: it can be read-only, write-only, or read-write. It can also allow arbitrary random access (seeking forwards or backwards to any location), or only sequential access (for example in the case of a socket or pipe).

All streams are careful about the type of data you give to them. For example giving a *str* object to the `write()` method of a binary stream will raise a *TypeError*. So will giving a *bytes* object to the `write()` method of a text stream.

Alterado na versão 3.3: Operations that used to raise *IOError* now raise *OSError*, since *IOError* is now an alias of *OSError*.

Text I/O

Text I/O expects and produces *str* objects. This means that whenever the backing store is natively made of bytes (such as in the case of a file), encoding and decoding of data is made transparently as well as optional translation of platform-specific newline characters.

The easiest way to create a text stream is with *open()*, optionally specifying an encoding:

```
f = open("myfile.txt", "r", encoding="utf-8")
```

In-memory text streams are also available as *StringIO* objects:

```
f = io.StringIO("some initial text data")
```

The text stream API is described in detail in the documentation of *TextIOBase*.

Binary I/O

Binary I/O (also called *buffered I/O*) expects *bytes-like objects* and produces *bytes* objects. No encoding, decoding, or newline translation is performed. This category of streams can be used for all kinds of non-text data, and also when manual control over the handling of text data is desired.

The easiest way to create a binary stream is with *open()* with `'b'` in the mode string:

```
f = open("myfile.jpg", "rb")
```

In-memory binary streams are also available as *BytesIO* objects:

```
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

The binary stream API is described in detail in the docs of *BufferedIOBase*.

Other library modules may provide additional ways to create text or binary streams. See *socket.socket.makefile()* for example.

Raw I/O

Raw I/O (also called *unbuffered I/O*) is generally used as a low-level building-block for binary and text streams; it is rarely useful to directly manipulate a raw stream from user code. Nevertheless, you can create a raw stream by opening a file in binary mode with buffering disabled:

```
f = open("myfile.jpg", "rb", buffering=0)
```

The raw stream API is described in detail in the docs of *RawIOBase*.

16.2.2 High-level Module Interface

`io.`**`DEFAULT_BUFFER_SIZE`**

An int containing the default buffer size used by the module's buffered I/O classes. *open()* uses the file's blksize (as obtained by *os.stat()*) if possible.

`io.`**`open`**(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)

This is an alias for the builtin *open()* function.

exception io.**BlockingIOError**
This is a compatibility alias for the builtin *BlockingIOError* exception.

exception io.**UnsupportedOperation**
An exception inheriting *OSError* and *ValueError* that is raised when an unsupported operation is called on a stream.

In-memory streams

It is also possible to use a *str* or *bytes-like object* as a file for both reading and writing. For strings *StringIO* can be used like a file opened in text mode. *BytesIO* can be used like a file opened in binary mode. Both provide full read-write capabilities with random access.

Ver também:

sys contains the standard IO streams: *sys.stdin*, *sys.stdout*, and *sys.stderr*.

16.2.3 hierarquia de classe

The implementation of I/O streams is organized as a hierarchy of classes. First *abstract base classes* (ABCs), which are used to specify the various categories of streams, then concrete classes providing the standard stream implementations.

> **Nota:** The abstract base classes also provide default implementations of some methods in order to help implementation of concrete stream classes. For example, *BufferedIOBase* provides unoptimized implementations of `readinto()` and *readline()*.

At the top of the I/O hierarchy is the abstract base class *IOBase*. It defines the basic interface to a stream. Note, however, that there is no separation between reading and writing to streams; implementations are allowed to raise *UnsupportedOperation* if they do not support a given operation.

The *RawIOBase* ABC extends *IOBase*. It deals with the reading and writing of bytes to a stream. *FileIO* subclasses *RawIOBase* to provide an interface to files in the machine's file system.

The *BufferedIOBase* ABC deals with buffering on a raw byte stream (*RawIOBase*). Its subclasses, *BufferedWriter*, *BufferedReader*, and *BufferedRWPair* buffer streams that are readable, writable, and both readable and writable. *BufferedRandom* provides a buffered interface to random access streams. Another *BufferedIOBase* subclass, *BytesIO*, is a stream of in-memory bytes.

The *TextIOBase* ABC, another subclass of *IOBase*, deals with streams whose bytes represent text, and handles encoding and decoding to and from strings. *TextIOWrapper*, which extends it, is a buffered text interface to a buffered raw stream (*BufferedIOBase*). Finally, *StringIO* is an in-memory stream for text.

Argument names are not part of the specification, and only the arguments of *open()* are intended to be used as keyword arguments.

The following table summarizes the ABCs provided by the *io* module:

ABC	Inherits	Stub Methods	Mixin Methods and Properties
IOBase		`fileno`, `seek`, and `truncate`	`close`, `closed`, `__enter__`, `__exit__`, `flush`, `isatty`, `__iter__`, `__next__`, `readable`, `readline`, `readlines`, `seekable`, `tell`, `writable`, and `writelines`
RawIOBase	*IOBase*	`readinto` and `write`	Inherited *IOBase* methods, `read`, and `readall`
BufferedIOBase	*IOBase*	`detach`, `read`, `read1`, and `write`	Inherited *IOBase* methods, `readinto`, and `readinto1`
TextIOBase	*IOBase*	`detach`, `read`, `readline`, and `write`	Inherited *IOBase* methods, `encoding`, `errors`, and `newlines`

I/O Base Classes

class io.**IOBase**

The abstract base class for all I/O classes, acting on streams of bytes. There is no public constructor.

This class provides empty abstract implementations for many methods that derived classes can override selectively; the default implementations represent a file that cannot be read, written or seeked.

Even though *IOBase* does not declare `read()` or `write()` because their signatures will vary, implementations and clients should consider those methods part of the interface. Also, implementations may raise a *ValueError* (or *UnsupportedOperation*) when operations they do not support are called.

The basic type used for binary data read from or written to a file is *bytes*. Other *bytes-like objects* are accepted as method arguments too. Text I/O classes work with *str* data.

Note that calling any method (even inquiries) on a closed stream is undefined. Implementations may raise *ValueError* in this case.

IOBase (and its subclasses) supports the iterator protocol, meaning that an *IOBase* object can be iterated over yielding the lines in a stream. Lines are defined slightly differently depending on whether the stream is a binary stream (yielding bytes), or a text stream (yielding character strings). See *readline()* below.

IOBase is also a context manager and therefore supports the `with` statement. In this example, *file* is closed after the `with` statement's suite is finished—even if an exception occurs:

```
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

IOBase provides these data attributes and methods:

close()

Flush and close this stream. This method has no effect if the file is already closed. Once the file is closed, any operation on the file (e.g. reading or writing) will raise a *ValueError*.

As a convenience, it is allowed to call this method more than once; only the first call, however, will have an effect.

closed

`True` if the stream is closed.

fileno()
Return the underlying file descriptor (an integer) of the stream if it exists. An *OSError* is raised if the IO object does not use a file descriptor.

flush()
Flush the write buffers of the stream if applicable. This does nothing for read-only and non-blocking streams.

isatty()
Return `True` if the stream is interactive (i.e., connected to a terminal/tty device).

readable()
Return `True` if the stream can be read from. If `False`, `read()` will raise *OSError*.

readline(*size=-1*)
Read and return one line from the stream. If *size* is specified, at most *size* bytes will be read.

The line terminator is always `b'\n'` for binary files; for text files, the *newline* argument to *open()* can be used to select the line terminator(s) recognized.

readlines(*hint=-1*)
Read and return a list of lines from the stream. *hint* can be specified to control the number of lines read: no more lines will be read if the total size (in bytes/characters) of all lines so far exceeds *hint*.

Note that it's already possible to iterate on file objects using `for line in file: ...` without calling `file.readlines()`.

seek(*offset*, *whence=SEEK_SET*)
Change the stream position to the given byte *offset*. *offset* is interpreted relative to the position indicated by *whence*. The default value for *whence* is `SEEK_SET`. Values for *whence* are:

- `SEEK_SET` or `0` – start of the stream (the default); *offset* should be zero or positive
- `SEEK_CUR` or `1` – current stream position; *offset* may be negative
- `SEEK_END` or `2` – end of the stream; *offset* is usually negative

Return the new absolute position.

Novo na versão 3.1: The `SEEK_*` constants.

Novo na versão 3.3: Some operating systems could support additional values, like `os.SEEK_HOLE` or `os.SEEK_DATA`. The valid values for a file could depend on it being open in text or binary mode.

seekable()
Return `True` if the stream supports random access. If `False`, *seek()*, *tell()* and *truncate()* will raise *OSError*.

tell()
Return the current stream position.

truncate(*size=None*)
Resize the stream to the given *size* in bytes (or the current position if *size* is not specified). The current stream position isn't changed. This resizing can extend or reduce the current file size. In case of extension, the contents of the new file area depend on the platform (on most systems, additional bytes are zero-filled). The new file size is returned.

Alterado na versão 3.5: Windows will now zero-fill files when extending.

writable()
Return `True` if the stream supports writing. If `False`, `write()` and *truncate()* will raise *OSError*.

writelines(*lines*)
Write a list of lines to the stream. Line separators are not added, so it is usual for each of the lines provided to have a line separator at the end.

__del__()
Prepare for object destruction. *IOBase* provides a default implementation of this method that calls the instance's *close()* method.

class io.**RawIOBase**
Base class for raw binary I/O. It inherits *IOBase*. There is no public constructor.

Raw binary I/O typically provides low-level access to an underlying OS device or API, and does not try to encapsulate it in high-level primitives (this is left to Buffered I/O and Text I/O, described later in this page).

In addition to the attributes and methods from *IOBase*, *RawIOBase* provides the following methods:

read(*size=-1*)
Read up to *size* bytes from the object and return them. As a convenience, if *size* is unspecified or -1, all bytes until EOF are returned. Otherwise, only one system call is ever made. Fewer than *size* bytes may be returned if the operating system call returns fewer than *size* bytes.

If 0 bytes are returned, and *size* was not 0, this indicates end of file. If the object is in non-blocking mode and no bytes are available, `None` is returned.

The default implementation defers to *readall()* and *readinto()*.

readall()
Read and return all the bytes from the stream until EOF, using multiple calls to the stream if necessary.

readinto(*b*)
Read bytes into a pre-allocated, writable *bytes-like object* *b*, and return the number of bytes read. For example, *b* might be a *bytearray*. If the object is in non-blocking mode and no bytes are available, `None` is returned.

write(*b*)
Write the given *bytes-like object*, *b*, to the underlying raw stream, and return the number of bytes written. This can be less than the length of *b* in bytes, depending on specifics of the underlying raw stream, and especially if it is in non-blocking mode. `None` is returned if the raw stream is set not to block and no single byte could be readily written to it. The caller may release or mutate *b* after this method returns, so the implementation should only access *b* during the method call.

class io.**BufferedIOBase**
Base class for binary streams that support some kind of buffering. It inherits *IOBase*. There is no public constructor.

The main difference with *RawIOBase* is that methods *read()*, *readinto()* and *write()* will try (respectively) to read as much input as requested or to consume all given output, at the expense of making perhaps more than one system call.

In addition, those methods can raise *BlockingIOError* if the underlying raw stream is in non-blocking mode and cannot take or give enough data; unlike their *RawIOBase* counterparts, they will never return `None`.

Besides, the *read()* method does not have a default implementation that defers to *readinto()*.

A typical *BufferedIOBase* implementation should not inherit from a *RawIOBase* implementation, but wrap one, like *BufferedWriter* and *BufferedReader* do.

BufferedIOBase provides or overrides these methods and attribute in addition to those from *IOBase*:

raw
The underlying raw stream (a *RawIOBase* instance) that *BufferedIOBase* deals with. This is not part of the *BufferedIOBase* API and may not exist on some implementations.

detach()
Separate the underlying raw stream from the buffer and return it.

After the raw stream has been detached, the buffer is in an unusable state.

Some buffers, like *BytesIO*, do not have the concept of a single raw stream to return from this method. They raise *UnsupportedOperation*.

Novo na versão 3.1.

read(*size=-1*)
Read and return up to *size* bytes. If the argument is omitted, `None`, or negative, data is read and returned until EOF is reached. An empty *bytes* object is returned if the stream is already at EOF.

If the argument is positive, and the underlying raw stream is not interactive, multiple raw reads may be issued to satisfy the byte count (unless EOF is reached first). But for interactive raw streams, at most one raw read will be issued, and a short result does not imply that EOF is imminent.

A *BlockingIOError* is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

read1([*size*])
Read and return up to *size* bytes, with at most one call to the underlying raw stream's *read()* (or *readinto()*) method. This can be useful if you are implementing your own buffering on top of a *BufferedIOBase* object.

If *size* is `-1` (the default), an arbitrary number of bytes are returned (more than zero unless EOF is reached).

readinto(*b*)
Read bytes into a pre-allocated, writable *bytes-like object* *b* and return the number of bytes read. For example, *b* might be a *bytearray*.

Like *read()*, multiple reads may be issued to the underlying raw stream, unless the latter is interactive.

A *BlockingIOError* is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

readinto1(*b*)
Read bytes into a pre-allocated, writable *bytes-like object* *b*, using at most one call to the underlying raw stream's *read()* (or *readinto()*) method. Return the number of bytes read.

A *BlockingIOError* is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

Novo na versão 3.5.

write(*b*)
Write the given *bytes-like object*, *b*, and return the number of bytes written (always equal to the length of *b* in bytes, since if the write fails an *OSError* will be raised). Depending on the actual implementation, these bytes may be readily written to the underlying stream, or held in a buffer for performance and latency reasons.

When in non-blocking mode, a *BlockingIOError* is raised if the data needed to be written to the raw stream but it couldn't accept all the data without blocking.

The caller may release or mutate *b* after this method returns, so the implementation should only access *b* during the method call.

Raw File I/O

class io.**FileIO**(*name*, *mode='r'*, *closefd=True*, *opener=None*)

FileIO represents an OS-level file containing bytes data. It implements the *RawIOBase* interface (and therefore the *IOBase* interface, too).

The *name* can be one of two things:

- a character string or *bytes* object representing the path to the file which will be opened. In this case closefd must be `True` (the default) otherwise an error will be raised.
- an integer representing the number of an existing OS-level file descriptor to which the resulting *FileIO* object will give access. When the FileIO object is closed this fd will be closed as well, unless *closefd* is set to `False`.

The *mode* can be `'r'`, `'w'`, `'x'` or `'a'` for reading (default), writing, exclusive creation or appending. The file will be created if it doesn't exist when opened for writing or appending; it will be truncated when opened for writing. *FileExistsError* will be raised if it already exists when opened for creating. Opening a file for creating implies writing, so this mode behaves in a similar way to `'w'`. Add a `'+'` to the mode to allow simultaneous reading and writing.

The `read()` (when called with a positive argument), `readinto()` and `write()` methods on this class will only make one system call.

A custom opener can be used by passing a callable as *opener*. The underlying file descriptor for the file object is then obtained by calling *opener* with (*name*, *flags*). *opener* must return an open file descriptor (passing *os.open* as *opener* results in functionality similar to passing `None`).

O recém criado arquivo é ref:*non-inheritable 1*.

See the *open()* built-in function for examples on using the *opener* parameter.

Alterado na versão 3.3: The *opener* parameter was added. The `'x'` mode was added.

Alterado na versão 3.4: O arquivo agora é não herdável.

In addition to the attributes and methods from *IOBase* and *RawIOBase*, *FileIO* provides the following data attributes:

mode
The mode as given in the constructor.

name
The file name. This is the file descriptor of the file when no name is given in the constructor.

Buffered Streams

Buffered I/O streams provide a higher-level interface to an I/O device than raw I/O does.

class io.**BytesIO**([*initial_bytes*])

A stream implementation using an in-memory bytes buffer. It inherits *BufferedIOBase*. The buffer is discarded when the *close()* method is called.

The optional argument *initial_bytes* is a *bytes-like object* that contains initial data.

BytesIO provides or overrides these methods in addition to those from *BufferedIOBase* and *IOBase*:

getbuffer()
Return a readable and writable view over the contents of the buffer without copying them. Also, mutating the view will transparently update the contents of the buffer:

```
>>> b = io.BytesIO(b"abcdef")
>>> view = b.getbuffer()
>>> view[2:4] = b"56"
>>> b.getvalue()
b'ab56ef'
```

Nota: As long as the view exists, the *BytesIO* object cannot be resized or closed.

Novo na versão 3.2.

getvalue()
Return *bytes* containing the entire contents of the buffer.

read1([*size*])
In *BytesIO*, this is the same as *read()*.

Alterado na versão 3.7: The *size* argument is now optional.

readinto1(*b*)
In *BytesIO*, this is the same as *readinto()*.

Novo na versão 3.5.

class io.**BufferedReader**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
A buffer providing higher-level access to a readable, sequential *RawIOBase* object. It inherits *BufferedIOBase*. When reading data from this object, a larger amount of data may be requested from the underlying raw stream, and kept in an internal buffer. The buffered data can then be returned directly on subsequent reads.

The constructor creates a *BufferedReader* for the given readable *raw* stream and *buffer_size*. If *buffer_size* is omitted, *DEFAULT_BUFFER_SIZE* is used.

BufferedReader provides or overrides these methods in addition to those from *BufferedIOBase* and *IOBase*:

peek([*size*])
Return bytes from the stream without advancing the position. At most one single read on the raw stream is done to satisfy the call. The number of bytes returned may be less or more than requested.

read([*size*])
Read and return *size* bytes, or if *size* is not given or negative, until EOF or if the read call would block in non-blocking mode.

read1([*size*])
Read and return up to *size* bytes with only one call on the raw stream. If at least one byte is buffered, only buffered bytes are returned. Otherwise, one raw stream read call is made.

Alterado na versão 3.7: The *size* argument is now optional.

class io.**BufferedWriter**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
A buffer providing higher-level access to a writeable, sequential *RawIOBase* object. It inherits *BufferedIOBase*. When writing to this object, data is normally placed into an internal buffer. The buffer will be written out to the underlying *RawIOBase* object under various conditions, including:

- when the buffer gets too small for all pending data;
- when *flush()* is called;
- when a `seek()` is requested (for *BufferedRandom* objects);
- when the *BufferedWriter* object is closed or destroyed.

The constructor creates a *BufferedWriter* for the given writeable *raw* stream. If the *buffer_size* is not given, it defaults to *DEFAULT_BUFFER_SIZE*.

BufferedWriter provides or overrides these methods in addition to those from *BufferedIOBase* and *IOBase*:

flush()
Force bytes held in the buffer into the raw stream. A *BlockingIOError* should be raised if the raw stream blocks.

write(*b*)
Write the *bytes-like object*, *b*, and return the number of bytes written. When in non-blocking mode, a *BlockingIOError* is raised if the buffer needs to be written out but the raw stream blocks.

class io.**BufferedRandom**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
A buffered interface to random access streams. It inherits *BufferedReader* and *BufferedWriter*, and further supports `seek()` and `tell()` functionality.

The constructor creates a reader and writer for a seekable raw stream, given in the first argument. If the *buffer_size* is omitted it defaults to *DEFAULT_BUFFER_SIZE*.

BufferedRandom is capable of anything *BufferedReader* or *BufferedWriter* can do.

class io.**BufferedRWPair**(*reader*, *writer*, *buffer_size=DEFAULT_BUFFER_SIZE*)
A buffered I/O object combining two unidirectional *RawIOBase* objects – one readable, the other writeable – into a single bidirectional endpoint. It inherits *BufferedIOBase*.

reader and *writer* are *RawIOBase* objects that are readable and writeable respectively. If the *buffer_size* is omitted it defaults to *DEFAULT_BUFFER_SIZE*.

BufferedRWPair implements all of *BufferedIOBase*'s methods except for *detach()*, which raises *UnsupportedOperation*.

> **Aviso:** *BufferedRWPair* does not attempt to synchronize accesses to its underlying raw streams. You should not pass it the same object as reader and writer; use *BufferedRandom* instead.

Text I/O

class io.**TextIOBase**
Base class for text streams. This class provides a character and line based interface to stream I/O. It inherits *IOBase*. There is no public constructor.

TextIOBase provides or overrides these data attributes and methods in addition to those from *IOBase*:

encoding
The name of the encoding used to decode the stream's bytes into strings, and to encode strings into bytes.

errors
The error setting of the decoder or encoder.

newlines
A string, a tuple of strings, or `None`, indicating the newlines translated so far. Depending on the implementation and the initial constructor flags, this may not be available.

buffer
The underlying binary buffer (a *BufferedIOBase* instance) that *TextIOBase* deals with. This is not part of the *TextIOBase* API and may not exist in some implementations.

detach()
Separate the underlying binary buffer from the *TextIOBase* and return it.

After the underlying buffer has been detached, the *TextIOBase* is in an unusable state.

Some *TextIOBase* implementations, like *StringIO*, may not have the concept of an underlying buffer and calling this method will raise *UnsupportedOperation*.

Novo na versão 3.1.

read(*size=-1*)
Read and return at most *size* characters from the stream as a single *str*. If *size* is negative or None, reads until EOF.

readline(*size=-1*)
Read until newline or EOF and return a single str. If the stream is already at EOF, an empty string is returned.

If *size* is specified, at most *size* characters will be read.

seek(*offset*, *whence=SEEK_SET*)
Change the stream position to the given *offset*. Behaviour depends on the *whence* parameter. The default value for *whence* is SEEK_SET.

- SEEK_SET or 0: seek from the start of the stream (the default); *offset* must either be a number returned by *TextIOBase.tell()*, or zero. Any other *offset* value produces undefined behaviour.
- SEEK_CUR or 1: "seek" to the current position; *offset* must be zero, which is a no-operation (all other values are unsupported).
- SEEK_END or 2: seek to the end of the stream; *offset* must be zero (all other values are unsupported).

Return the new absolute position as an opaque number.

Novo na versão 3.1: The SEEK_* constants.

tell()
Return the current stream position as an opaque number. The number does not usually represent a number of bytes in the underlying binary storage.

write(*s*)
Write the string *s* to the stream and return the number of characters written.

class io.**TextIOWrapper**(*buffer*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, *write_through=False*)
A buffered text stream over a *BufferedIOBase* binary stream. It inherits *TextIOBase*.

encoding gives the name of the encoding that the stream will be decoded or encoded with. It defaults to *locale.getpreferredencoding(False)*.

errors is an optional string that specifies how encoding and decoding errors are to be handled. Pass 'strict' to raise a *ValueError* exception if there is an encoding error (the default of None has the same effect), or pass 'ignore' to ignore errors. (Note that ignoring encoding errors can lead to data loss.) 'replace' causes a replacement marker (such as '?') to be inserted where there is malformed data. 'backslashreplace' causes malformed data to be replaced by a backslashed escape sequence. When writing, 'xmlcharrefreplace' (replace with the appropriate XML character reference) or 'namereplace' (replace with \N{...} escape sequences) can be used. Any other error handling name that has been registered with *codecs.register_error()* is also valid.

newline controls how line endings are handled. It can be None, '', '\n', '\r', and '\r\n'. It works as follows:

- When reading input from the stream, if *newline* is None, *universal newlines* mode is enabled. Lines in the input can end in '\n', '\r', or '\r\n', and these are translated into '\n' before being returned to the

caller. If it is `''`, universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.

- Ao gravar a saída no fluxo, se *newline* for `None`, quaisquer caracteres `'\n'` gravados serão traduzidos para o separador de linhas padrão do sistema, *os.linesep*. Se *newline* for `''` ou `'\n'`, nenhuma tradução ocorrerá. Se *newline* for um dos outros valores legais, qualquer caractere `'\n'` escrito será traduzido para a sequência especificada.

If *line_buffering* is `True`, `flush()` is implied when a call to write contains a newline character or a carriage return.

If *write_through* is `True`, calls to `write()` are guaranteed not to be buffered: any data written on the *TextIOWrapper* object is immediately handled to its underlying binary *buffer*.

Alterado na versão 3.3: The *write_through* argument has been added.

Alterado na versão 3.3: The default *encoding* is now `locale.getpreferredencoding(False)` instead of `locale.getpreferredencoding()`. Don't change temporary the locale encoding using *locale.setlocale()*, use the current locale encoding instead of the user preferred encoding.

TextIOWrapper provides these members in addition to those of *TextIOBase* and its parents:

line_buffering
Whether line buffering is enabled.

write_through
Whether writes are passed immediately to the underlying binary buffer.

Novo na versão 3.7.

reconfigure(**[, encoding][, errors][, newline][, line_buffering][, write_through]*)
Reconfigure this text stream using new settings for *encoding*, *errors*, *newline*, *line_buffering* and *write_through*.

Parameters not specified keep current settings, except `errors='strict'` is used when *encoding* is specified but *errors* is not specified.

It is not possible to change the encoding or newline if some data has already been read from the stream. On the other hand, changing encoding after write is possible.

This method does an implicit stream flush before setting the new parameters.

Novo na versão 3.7.

class io.**StringIO**(*initial_value=''*, *newline='\n'*)
An in-memory stream for text I/O. The text buffer is discarded when the *close()* method is called.

The initial value of the buffer can be set by providing *initial_value*. If newline translation is enabled, newlines will be encoded as if by *write()*. The stream is positioned at the start of the buffer.

The *newline* argument works like that of *TextIOWrapper*. The default is to consider only `\n` characters as ends of lines and to do no newline translation. If *newline* is set to `None`, newlines are written as `\n` on all platforms, but universal newline decoding is still performed when reading.

StringIO provides this method in addition to those from *TextIOBase* and its parents:

getvalue()
Return a `str` containing the entire contents of the buffer. Newlines are decoded as if by *read()*, although the stream position is not changed.

Exemplo de uso:

```
import io

output = io.StringIO()
output.write('First line.\n')
print('Second line.', file=output)

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

class io.**IncrementalNewlineDecoder**
A helper codec that decodes newlines for *universal newlines* mode. It inherits *codecs.IncrementalDecoder*.

16.2.4 Performance

This section discusses the performance of the provided concrete I/O implementations.

Binary I/O

By reading and writing only large chunks of data even when the user asks for a single byte, buffered I/O hides any inefficiency in calling and executing the operating system's unbuffered I/O routines. The gain depends on the OS and the kind of I/O which is performed. For example, on some modern OSes such as Linux, unbuffered disk I/O can be as fast as buffered I/O. The bottom line, however, is that buffered I/O offers predictable performance regardless of the platform and the backing device. Therefore, it is almost always preferable to use buffered I/O rather than unbuffered I/O for binary data.

Text I/O

Text I/O over a binary storage (such as a file) is significantly slower than binary I/O over the same storage, because it requires conversions between unicode and binary data using a character codec. This can become noticeable handling huge amounts of text data like large log files. Also, `TextIOWrapper.tell()` and `TextIOWrapper.seek()` are both quite slow due to the reconstruction algorithm used.

StringIO, however, is a native in-memory unicode container and will exhibit similar speed to *BytesIO*.

Multi-threading

FileIO objects are thread-safe to the extent that the operating system calls (such as `read(2)` under Unix) they wrap are thread-safe too.

Binary buffered objects (instances of *BufferedReader*, *BufferedWriter*, *BufferedRandom* and *BufferedRWPair*) protect their internal structures using a lock; it is therefore safe to call them from multiple threads at once.

TextIOWrapper objects are not thread-safe.

Reentrancy

Binary buffered objects (instances of *BufferedReader*, *BufferedWriter*, *BufferedRandom* and *BufferedRWPair*) are not reentrant. While reentrant calls will not happen in normal situations, they can arise from doing I/O in a *signal* handler. If a thread tries to re-enter a buffered object which it is already accessing, a *RuntimeError* is raised. Note this doesn't prohibit a different thread from entering the buffered object.

The above implicitly extends to text files, since the *open()* function will wrap a buffered object inside a *TextIOWrapper*. This includes standard streams and therefore affects the built-in function *print()* as well.

16.3 `time` — Acesso ao horário e conversões

Esse módulo provê várias funções relacionadas à tempo. Para funcionalidades relacionadas veja também os módulos *datetime* e *calendar*

Apesar desse módulo sempre estar disponível, nem todas as suas funções estão disponíveis em todas as plataformas. A maioria das funções definidas nesse módulo chamam funções da biblioteca da plataforma de C com mesmo nome. Pode ser útil consultar a documentação da plataforma, pois da semântica dessas funções variam a depender da plataforma.

A seguir, uma explicação de algumas terminologias e convenções.

- A *epoca* é o ponto onde o tempo começa e depende da plataforma. Em Unix, a época é 1ª de Janeiro de 1970, ás 00:00:00 (UTC). Para descobrir a época em alguma plataforma, veja em `time.gmtime(0)`
- O termo *segundos desde a época* refere-se ao número total de segundos decorrido desde a época, tipicamente excluindo-se os *segundos bissextos*. Segundos bissextos são excluídos desse total em todas as plataformas compatíveis com POSIX.
- As funções desse módulo podem não conseguir tratar de datas e horários antes da época ou muito distantes no futuro. O limite no futuro é determinado pela biblioteca C; para sistemas de 32 bit, geralmente é 2038.
- A função *strptime()* pode analisar anos de 2 dígitos quando é passado o código de formato `%y`. Quando anos de 2 dígitos são analisados, eles são convertidos de acordo com os padrões POSIX e ISO C: valores 69–99 são mapeados para 1969–199, e valores 0–68 são mapeados para 2000–2068.
- UTC é Coordinated Universal Time (antigamente conhecido como Greenwich Mean Time ou GMT). O acrônimo UTC não é um erro, mas um acordo entre inglês e francês.
- DST é Daylight Saving Time (Horário de Verão), um ajuste de fuso horário por (normalmente) uma hora durante parte do ano. As regras de Horário de Verão são mágicas (determinadas por leis locais) e podem mudar de ano a ano. A biblioteca C possui uma tabela contendo as regras locais (normalmente lidas de um arquivo de sistema por flexibilidade) e nesse contexto é a única fonte de Conhecimento Verdadeiro.
- A precisão de várias funções em tempo real podem ser menores do que o que pode estar sugerido pelas unidades nas quais seu valor ou argumento estão expressos. E.g. na maioria dos sistemas Unix, o relógio "conta" apenas 50 ou 100 vezes por segundo.
- Por outro lado, a precisão de *time()* e *sleep()* é melhor do que suas equivalentes Unix: Tempos são expressos como números em ponto flutuante, *time()* retorna o tempo mais preciso disponível (utilizando `gettimeofday()` do Unix, quando disponível) e *sleep()* irá aceitar qualquer tempo como uma fração não zero (`select()` do Unix é utilizada para implementar isto, quando disponível).
- O valor de tempo conforme retornado pelas *gmtime()*, *localtime()*, e *strptime()*, e aceito pelas *asctime()*, *mktime()* e *strftime()*, é uma sequência de 9 inteiros. Os valores retornados das *gmtime()*, *localtime()*, e *strptime()* também oferecem nomes de atributo para campos individuais.

  Veja *struct_time* para a descrição desses objetos.

Alterado na versão 3.3: O tipo *struct_time* foi estendido para prover os atributos `tm_gmtoff` e `tm_zone` quando a plataforma suporta os membros `struct tm` correspondentes

Alterado na versão 3.6: Os atributos `tm_gmtoff` e `tm_zone` da classe *struct_time* estão disponíveis em todas as plataformas agora.

- Utilize as seguintes funções para converter entre representações de tempo:

De	Para	Utilize
segundos desde a época	*struct_time* em UTC	*gmtime()*
segundos desde a época	*struct_time* em tempo local	*localtime()*
struct_time em UTC	segundos desde a época	*calendar.timegm()*
struct_time em tempo local	segundos desde a época	*mktime()*

16.3.1 Funções

`time.`**`asctime`**`([t])`
Convert a tuple or *struct_time* representing a time as returned by *gmtime()* or *localtime()* to a string of the following form: `'Sun Jun 20 23:21:05 1993'`. If *t* is not provided, the current time as returned by *localtime()* is used. Locale information is not used by *asctime()*.

Nota: Diferentemente da função em C de mesmo nome, *asctime()* não adiciona uma nova linha em seguida.

`time.`**`clock`**`()`
On Unix, return the current processor time as a floating point number expressed in seconds. The precision, and in fact the very definition of the meaning of "processor time", depends on that of the C function of the same name.

On Windows, this function returns wall-clock seconds elapsed since the first call to this function, as a floating point number, based on the Win32 function `QueryPerformanceCounter()`. The resolution is typically better than one microsecond.

Deprecated since version 3.3, will be removed in version 3.8: The behaviour of this function depends on the platform: use *perf_counter()* or *process_time()* instead, depending on your requirements, to have a well defined behaviour.

`time.`**`pthread_getcpuclockid`**`(`*thread_id*`)`
Retorna o *clk_id* do relógio de tempo de CPU específico da thread para a *thread_id* especificada.

Utilize a *threading.get_ident()* ou o atributo *ident* dos objetos *threading.Thread* para obter um valor adequado para *thread_id*.

Aviso: Passando um *thread_id* inválido ou expirado pode resultar em um comportamento indefinido, como por exemplo falha de segmentação.

Disponibilidade: Unix (veja a página man para *pthread_getcpuclockid(3)* para mais informações).

Novo na versão 3.7.

`time.`**`clock_getres`**`(`*clk_id*`)`
Retorna a resolução (precisão) do relógio *clk_id* especificado. Confira *Constantes de ID de Relógio* para uma lista de valores aceitos para *clk_id*

Disponibilidade: Unix.

Novo na versão 3.3.

time.**clock_gettime**(*clk_id*) → float
Retorna o tempo to relógio *clk_id* especificado. Confira *Constantes de ID de Relógio* para uma lista de valores aceitos para *clk_id*.

Disponibilidade: Unix.

Novo na versão 3.3.

time.**clock_gettime_ns**(*clk_id*) → int
Similar à `clock_gettime()`, mas retorna o tempo em nanossegundos.

Disponibilidade: Unix.

Novo na versão 3.7.

time.**clock_settime**(*clk_id*, *time: float*)
Define o tempo do relógio *clk_id* especificado. Atualmente, `CLOCK_REALTIME` é o único valor aceito para *clk_id*.

Disponibilidade: Unix.

Novo na versão 3.3.

time.**clock_settime_ns**(*clk_id*, *time: int*)
Similar à `clock_settime()`, mas define o tempo em nanossegundos.

Disponibilidade: Unix.

Novo na versão 3.7.

time.**ctime**([*secs*])
Convert a time expressed in seconds since the epoch to a string representing local time. If *secs* is not provided or `None`, the current time as returned by `time()` is used. `ctime(secs)` is equivalent to `asctime(localtime(secs))`. Locale information is not used by `ctime()`.

time.**get_clock_info**(*name*)
Obtém informação do relógio específico como um objeto espaço de nomes. Nomes de relógios suportados e as funções correspondentes para ler seus valores são:

- `'clock'`: `time.clock()`
- `'monotonic'`: `time.monotonic()`
- `'perf_counter'`: `time.perf_counter()`
- `'process_time'`: `time.process_time()`
- `'thread_time'`: `time.thread_time()`
- `'time'`: `time.time()`

O resultado possui os seguintes atributos:

- *adjustable*: `True` se o relógio pode ser alterado automaticamente (e.g. por um daemon NTP) ou manualmente por um administrador do sistema, `False` se contrário
- *implementation*: O nome da função C subjacente utilizada para obter o valor do relógio. Confira *Constantes de ID de Relógio* para valores possíveis.
- *monotonic*: `True` se o relógio não pode retornar a valores anteriores, backward, `False` contrário
- *resolution*: A resolução do relógio em segundos (`float`)

Novo na versão 3.3.

time.**gmtime**([*secs*])
Converte um tempo expresso em segundos desde a época para uma `struct_time` em UTC onde o sinalizador de horário de verão é sempre zero. Se *secs* não é fornecido ou `None`, o tempo atual como retornado por `time()`

é utilizado. Frações de segundo são ignoradas. Veja acima para uma descrição do objeto *struct_time*. Veja *calendar.timegm()* para o inverso desta função.

time.**localtime**([*secs*])
Como *gmtime()*, mas converte para o tempo local. Se *secs* não é fornecido ou *None*, o tempo atual como retornado por *time()* é utilizado. O sinalizador de horário de verão é definido como `1` quando o Horário de verão for aplicável para o tempo fornecido.

time.**mktime**(*t*)
Esta é a função inversa de *localtime()*. Seu argumento é a *struct_time* ou uma 9-tupla (sendo o sinalizador de horário de verão necessário; utilize `-1` como sinalizador de horário de verão quando este for desconhecido) que expressa o tempo em tempo *local*, não UTC. Retorna um número em ponto flutuante, para ter compatibilidade com *time()*. Se o valor de entrada não puder ser representado como um tempo válido, ou *OverflowError* ou *ValueError* serão levantadas (o que irá depender se o valor inválido é capturado pelo Python ou por bibliotecas C subjacentes). A data mais recente para qual um tempo pode ser gerado é dependente da plataforma.

time.**monotonic**() → float
Return the value (in fractional seconds) of a monotonic clock, i.e. a clock that cannot go backwards. The clock is not affected by system clock updates. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid.

Novo na versão 3.3.

Alterado na versão 3.5: A função agora é sempre disponível e sempre de todo o sistema.

time.**monotonic_ns**() → int
Similar à *monotonic()*, mas retorna tempo em nanossegundos.

Novo na versão 3.7.

time.**perf_counter**() → float
Return the value (in fractional seconds) of a performance counter, i.e. a clock with the highest available resolution to measure a short duration. It does include time elapsed during sleep and is system-wide. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid.

Novo na versão 3.3.

time.**perf_counter_ns**() → int
Semelhante à *perf_counter()*, mas retorna o tempo em nanossegundos.

Novo na versão 3.7.

time.**process_time**() → float
Return the value (in fractional seconds) of the sum of the system and user CPU time of the current process. It does not include time elapsed during sleep. It is process-wide by definition. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid.

Novo na versão 3.3.

time.**process_time_ns**() → int
Semelhante à *process_time()*, mas retorna o tempo em nanossegundos.

Novo na versão 3.7.

time.**sleep**(*secs*)
Execução suspensa da thread chamada para o número de segundos dado. O argumento pode ser um número em ponto flutuante para indicar um tempo de sono mais preciso. O tempo de suspensão atual pode ser menor do que o solicitado porque qualquer sinal capturado irá terminar *sleep()* seguindo a execução da rotina de captura daquele sinal. Além disso, o tempo de suspensão pode ser maior do que o solicitado por uma quantidade arbitrária devido ao agendamento de outra atividade no sistema.

Alterado na versão 3.5: A função agora dorme por pelo menos *secs* mesmo se o sono é interrompido por um sinal, exceto se o tratador de sinal levanta uma exceção (veja **PEP 475** para a explicação).

time.**strftime**(*format*[, *t*])

Converte a tupla ou `struct_time` representando um tempo como retornado por `gmtime()` ou `localtime()` para uma string como especificado pelo argumento *format*. Se *t* não é fornecido, o tempo atual como retornado pela `localtime()` é utilizado. *format* deve ser uma string. `ValueError` é levantado se qualquer campo em *t* está fora do intervalo permitido.

0 é um argumento legal para qualquer posição na tupla de tempo; se é normalmente ilegal, o valor é formado a um valor correto.

As diretivas a seguir podem ser incorporadas na string *format*. Elas estão mostradas sem os campos de comprimento e especificação de precisão opcionais, e estão substituídos pelos caracteres indicados no resultado da `strftime()`:

Diretiva	Significado	Notas
`%a`	Nome abreviado do dia da semana da localidade.	
`%A`	Nome completo do dia da semana da localidade.	
`%b`	Nome abreviado do mês da localidade.	
`%B`	Nome completo do mês da localidade.	
`%c`	Representação de data e hora apropriada da localidade.	
`%d`	Dia do mês como um número decimal [01,31].	
`%H`	Hora (relógio 24 horas) como um número decimal [00,23].	
`%I`	Hora (relógio 12 horas) como um número decimal [01,12].	
`%j`	Dia do ano como um número decimal [001,366].	
`%m`	Mês como um número decimal [01,12].	
`%M`	Minuto como um número decimal [00,59].	
`%p`	Equivalente da localidade a AM ou PM.	(1)
`%S`	Segundo como um número decimal [00,61].	(2)
`%U`	Número da semana do ano (domingo como primeiro dia da semana) como um número decimal [00,53]. Todos os dias em um ano novo que precedem o primeiro domingo são considerados como estando na semana 0.	(3)
`%w`	Dia da semana como um número decimal [0(Domingo),6]	
`%W`	Número da semana do ano (segunda-feira como o primeiro dia da semana) como um número decimal [00,53]. Todos os dias do ano que precedem o primeiro domingo serão considerados como estando na semana 0.	(3)
`%x`	Representação de data apropriada de localidade.	
`%X`	Representação de hora apropriada da localidade.	
`%y`	Ano sem século como um número decimal [00,99].	
`%Y`	Ano com o século como um número decimal.	
`%z`	Deslocamento de fuso horário indicando uma diferença de tempo positiva ou negativa de UTC/GMT formatado como +HHMM ou -HHMM, onde H representa os dígitos decimais de hora e M representa os dígitos decimais de minuto [-23:59, +23:59].	
`%Z`	Nome do fuso horário (nenhum caractere se nenhum fuso horário existe).	
`%%`	Um caractere literal `'%'`.	

Notas:

(1) Quando utilizado com a função `strptime()`, a diretiva `%p` apenas afeta a saída do campo se a diretiva `%I` é utilizada para analisar a hora.

(2) O intervalo é realmente `0` até `61`; o valor `60` é válido em timestamps representando segundos bissextos e o

valor `61` é suportado por razões históricas.

(3) Quando utilizado co a função *`strptime()`*, `%U` e `%W` são utilizados em cálculos apenas quando o dia da semana e ano são especificados.

Veja este exemplo, um formato para datas compatível com as especificações dos padrões de e-mail **RFC 2822**.[1]

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

Diretivas adicionais podem ser suportadas por algumas plataformas, mas apenas as listadas aqui possuem significado padronizado por ANSI C. Para ver a lista completa de códigos de formato suportados na sua plataforma, consulte a documentação *`strftime(3)`*.

Em algumas plataformas, um campo adicional de comprimento e especificação de precisão podem seguir imediatamente após `'%'` como uma diretiva da seguinte ordem; isto também não é portátil. O campo comprimento normalmente é 2 exceto para `%j` quando é 3.

`time.`**`strptime`**(*string*[, *format*])

Analisa a string representando um tempo de acordo com um formato. O valor retornado é um *`struct_time`* como retornado por *`gmtime()`* ou *`localtime()`*.

O parâmetro *format* utiliza as mesmas diretivas das utilizadas por *`strftime()`*; é definido por padrão para `"%a %b %d %H:%M:%S %Y"` que corresponde com a formatação retornada por *`ctime()`*. Se *string* não puder ser analisada de acordo com *format*, ou se possui excesso de dados após analisar, *`ValueError`* é levantado. Os valores padrão utilizados para preencher quaisquer dados faltantes quando valores mais precisos não puderem ser inferidos são `(1900, 1, 1, 0, 0, 0, 0, 1, -1)`. Ambos *string* e *format* devem ser strings.

Por exemplo:

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0,
                 tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
```

Suporte para a diretiva `%Z` é baseado nos valores contidos em `tzname` e se `daylight` é verdade. Por causa disso, é específico de plataforma exceto por reconhecer UTC e GMT, que são sempre conhecidos (e considerados fuso horários sem horários de verão).

Apenas as diretivas especificadas na documentação são suportadas. Como `strftime()` é implementada por plataforma, esta pode apenas as vezes oferecer mais diretivas do que as listadas aqui. Mas `strptime()` é independente de quaisquer plataformas e portanto não necessariamente suporta todas as diretivas disponíveis que não estão documentadas como suportadas.

`class` `time.`**`struct_time`**

O tipo da sequência de valor de tempo retornado por *`gmtime()`*, *`localtime()`*, e *`strptime()`*. É um objeto com uma interface *named tuple*: os valores podem ser acessados por um índice e por um nome de atributo. Os seguintes valores estão presentes:

[1] A utilização de `%Z` está descontinuada, mas o escape `%z` que expande para um deslocamento hora/minuto preferido não é suportado por todas as bibliotecas ANSI C. Além disso, a leitura do padrão original **RFC 822** de 1982 mostra que este pede um ano com dois dígitos (%y em vez de %Y), mas a prática consolidou a utilização de anos com 4 dígitos mesmo antes dos anos 2000. Após isso, o **RFC 822** se tornou obsoleto e o ano de 4 dígitos foi primeiro recomendado por **RFC 1123** e depois obrigatório por **RFC 2822**.

Index	Atributo	Valores
0	`tm_year`	(por exemplo, 1993)
1	`tm_mon`	intervalo [1,12]
2	`tm_mday`	intervalo [1,31]
3	`tm_hour`	intervalo [0,23]
4	`tm_min`	intervalo [0,59]
5	`tm_sec`	intervalo [0,61]; veja **(2)** na descrição de *`strftime()`*
6	`tm_wday`	intervalo [0,6], segunda-feira é 0
7	`tm_yday`	intervalo [1, 366]
8	`tm_isdst`	0, 1 ou -1; veja abaixo
N/D	`tm_zone`	abreviação do nome do fuso horário
N/D	`tm_gmtoff`	deslocamento a leste de UTC em segundos

Note que diferentemente da estrutura C, o valor do mês é um intervalo [1,12] e não [0,11].

Em chamadas para *`mktime()`*, `tm_isdst` pode ser definido como 1 quando o horário de verão estiver em efeito, e 0 quando não. Um valor de -1 indica que esta informação não é conhecida, e geralmente resultará no preenchimento do estado correto.

Quando uma tupla com comprimento incorreto é passada para uma função que espera por um *`struct_time`*, ou por possuir elementos do tipo errado, um *`TypeError`* é levantado.

`time.`**`time`**`()` → float
Retorna o tempo em segundos desde a era como um número em ponto flutuante. A data específica da era e a manipulação de *segundos bissextos* são dependentes da plataforma. Em Windows e na maioria dos sistemas Unix, a era é 1 de janeiro de 1970, 00:00:00 (UTC)

Note que mesmo o tempo sendo retornado sempre como um número em ponto flutuante, nem todos os sistemas fornecem o tempo com precisão melhor que 1 segundo. Enquanto esta função normalmente retorna valores não decrescentes, pode retornar valores menores do que os de uma chamada anterior se o relógio do sistema foi redefinido entre duas chamadas.

O número retornado por *`time()`* pode ser convertido a um formato de tempo mais comum (i.e. ano, mês, dia, hora etc…) em UTC por passá-lo para a função *`gmtime()`* ou em tempo local por passar para a função *`localtime()`*. Em ambos os casos, o objeto *`struct_time`* é retornado, por onde os componentes de data do calendário podem ser acessados ou atribuídos.

`time.`**`thread_time`**`()` → float
Return the value (in fractional seconds) of the sum of the system and user CPU time of the current thread. It does not include time elapsed during sleep. It is thread-specific by definition. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls in the same thread is valid.

Disponibilidade: Windows, Linux, sistemas Unix systems suportando `CLOCK_THREAD_CPUTIME_ID`.

Novo na versão 3.7.

`time.`**`thread_time_ns`**`()` → int
Semelhante à *`thread_time()`*, mas retorna o tempo em nanossegundos.

Novo na versão 3.7.

`time.`**`time_ns`**`()` → int
Semelhante à *`time()`*, mas retorna o tempo como um número inteiro de nanossegundos desde a *era*.

Novo na versão 3.7.

`time.`**`tzset`**`()`
Redefine as regras de conversão utilizadas pelas rotinas da biblioteca. A variável de ambiente `TZ` especifica como isto é feito. Também irá redefinir as variáveis `tzname` (da variável de ambiente `TZ`), `timezone` (segundos sem

horário de verão a oeste de UTC), `altzone` (segundos com horário de verão a oeste de UTC) e `daylight` (para 0 se este fuso horário não possui nenhuma regra de horário de verão, ou diferente de zero se há um tempo, no presente, passado ou futuro quando o horário de verão se aplica).

Disponibilidade: Unix.

Nota: Embora em vários casos, alterar a variável de sistema `TZ` pode afetar a saída de funções como *`localtime()`* sem chamar *`tzset()`*, este comportamento não deve ser confiado.

A variável de sistema `TZ` não deve conter espaços em branco.

O formato padrão da variável de sistema `TZ` é (espaços foram adicionados por motivos de clareza):

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

Onde os componentes são:

`std` e `dst` Três ou mais alfanuméricos fornecendo a abreviação do fuso horário. Estes serão propagados para time.tzname

`offset` O deslocamento tem a forma: `± hh[:mm[:ss]]`. Isso indica que o valor adicionado adicionou o horário local para chegar a UTC. Se precedido por um '-', o fuso horário está a leste do Meridiano Primário; do contrário, está a oeste. Se nenhum deslocamento segue o horário de verão, o tempo no verão é assumido como estando uma hora a frente do horário padrão.

`start[/time], end[/time]` Indica quando mudar e voltar do Horário de Verão. O formato das datas de início e fim é um dos seguintes:

`Jn` I dia juliano *n* (1 <= *n* <= 365). Os dias bissextos não são contados, então, em todos os anos, 28 de fevereiro é o dia 59 e 1 de março é o dia 60.

`n` O dia juliano baseado em zero (0 <= *n* <= 365). Dias bissextos são contados, e é possível fazer referência a 29 de fevereiro.

`Mm.n.d` O *d*-ésimo dia (0 <= *d* <= 6) da semana *n* do mês *m* do ano (1 <= *n* <= 5, 1 <= *m* <= 12, onde semana 5 significa "o último dia *d* no mês *m*" que pode ocorrer tanto na quarta como quinta semana). Semana 1 é a primeira semana na qual o *d*-ésimo dia ocorre. Dia zero é o domingo.

`time` tem o mesmo formato que `offset`, exceto que nenhum sinal no início é permitido ('-' ou '+'). O padrão, se o tempo não é dado, é 02:00:00.

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

Em muitos sistemas Unix (incluindo *BSD, Linux, Solaris, e Darwin), é mais conveniente utilizar o banco de dados de informação de fuso do sistema (*`tzfile(5)`*) para especificar as regras de fuso horário. Para fazer isso, defina a variável de sistema `TZ` ao path do arquivo de dados requerido de fuso horários, relativo à raiz do banco de dados de fuso horário 'zoneinfo' do sistema, geralmente encontrado em `/usr/share/zoneinfo`. Por exemplo, `'US/Eastern'`, `'Australia/Melbourne'`, `'Egypt'` ou `'Europe/Amsterdam'`.

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
```

(continua na próxima página)

(continuação da página anterior)

```
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

16.3.2 Constantes de ID de Relógio

Essas constantes são utilizadas como parâmetros para *clock_getres()* e *clock_gettime()*.

time.**CLOCK_BOOTTIME**
Idêntica a *CLOCK_MONOTONIC*, exceto por também incluir qualquer tempo que o sistema está suspenso.

Isto permite que aplicações recebam um relógio monotônico consciente suspenso sem precisar lidar com as complicações de *CLOCK_REALTIME*, que pode conter descontinuidades se o tempo é alterado utilizando `settimeofday()` ou algo semelhante.

Availability: Linux 2.6.39 ou posterior.

Novo na versão 3.7.

time.**CLOCK_HIGHRES**
O Solaris OS possui um timer `CLOCK_HIGHRES` que tenta utilizar recursos otimizados do hardware, e pode fornecer resolução perto de nanossegundos. `CLOCK_HIGHRES` é o relógio nanoajustável de alta resolução.

Disponibilidade: Solaris.

Novo na versão 3.3.

time.**CLOCK_MONOTONIC**
Relógio que não pode ser definido e representa um tempo monotônico desde um ponto de início não especificado.

Disponibilidade: Unix.

Novo na versão 3.3.

time.**CLOCK_MONOTONIC_RAW**
Semelhante à *CLOCK_MONOTONIC*, mas fornece acesso a um tempo bruto baseado em hardware que não está sujeito a ajustes NTP.

Disponibilidade: Linux 2.6.28 ou mais recente, macOS 10.12 ou mais recente.

Novo na versão 3.3.

time.**CLOCK_PROCESS_CPUTIME_ID**
Timer de alta resolução por processo no CPU.

Disponibilidade: Unix.

Novo na versão 3.3.

time.**CLOCK_PROF**
Timer de alta resolução por processo no CPU.

Disponibilidade: FreeBSD, NetBSD 7 ou posterior, OpenBSD.

Novo na versão 3.7.

time.**CLOCK_THREAD_CPUTIME_ID**
Relógio de tempo de CPU específico a thread.

Disponibilidade: Unix.

Novo na versão 3.3.

time.**CLOCK_UPTIME**
Tempo cujo valor absoluto é o tempo que o sistema está sendo executado e não suspenso, fornecendo medidas de tempo de atividade precisas, tanto em valor absoluto quanto intervalo.

Disponibilidade: FreeBSD, OpenBSD 5.5 ou posterior.

Novo na versão 3.7.

A constante a seguir é o único parâmetro que pode ser enviado para *clock_settime()*.

time.**CLOCK_REALTIME**
Relógio em tempo real de todo o sistema. Definições deste relógio requerem privilégios apropriados.

Disponibilidade: Unix.

Novo na versão 3.3.

16.3.3 Constantes de Fuso Horário

time.**altzone**
O deslocamento do fuso horário DST local, em segundos a oeste de UTC, se algum for fornecido. É negativo se o fuso horário DST local está a leste de UTC (como na Europa Ocidental, incluindo o Reino Unido). Somente utilize se `daylight` for diferente de zero. Veja a nota abaixo.

time.**daylight**
Diferente de zero se um fuso horário DST é definido. Veja nota abaixo.

time.**timezone**
O deslocamento para o fuso horário local (não DST), em segundos a oeste de UTC (negativo na maior parte da Europa Ocidental, positivo nos Estados Unidos e Brasil, zero no Reino Unido). Ver nota abaixo.

time.**tzname**
A tupla de duas strings: A primeira é o nome do fuso horário local não DST, a segunda é o nome do fuso horário local DST. Se nenhum fuso horário DST for definido, a segunda string é usada. Veja nota abaixo.

Nota: Para as constantes de Fuso Horário acima (*altzone*, *daylight*, *timezone*, e *tzname*), o valor é determinado pelas regras de fuso horário em efeito no módulo de carregamento de tempo ou a última vez que *tzset()* é chamada e pode estar incorreto para tempos no passado. É recomendado utilizar os resultados `tm_gmtoff` e `tm_zone` da *localtime()* para obter informação de fuso horário.

Ver também:

Módulo *datetime* Mais interfaces orientada a objetos para datas e tempos.

Módulo *locale* Serviços de internacionalização. A configuração de localidade afeta a interpretação de muitos especificadores de formato em *strftime()* e *strptime()*.

Módulo *calendar* Funções gerais relacionadas a calendários. *timegm()* é a função inversa de *gmtime()* deste módulo.

16.4 argparse — Parser para opções de linha de comando, argumentos e subcomandos

Novo na versão 3.2.

Código Fonte: Lib/argparse.py

Tutorial

Esta página contém informações da API de Referência. Para uma introdução mais prática para o parser de linha de comando Python, acesse argparse tutorial.

O módulo *argparse* torna fácil a escrita de interfaces de linha de comando amigáveis. O programa define quais argumentos são necessários e *argparse* descobrirá como analisá-lo e interpretá-los a partir do *sys.argv*. O módulo *argparse* também gera automaticamente o texto ajuda, mensagens de uso e error emitidos quando o usuário prover argumentos inválidos para o programa.

16.4.1 Exemplo

O código a seguir é um programa Python que recebe uma lista de inteiros e apresenta a soma ou o máximo:

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

Assumindo que o código Python acima está salvo em um arquivo chamado `prog.py`, pode-se executá-lo pela linha de comando e obter mensagens de ajuda úteis:

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
 N           an integer for the accumulator

optional arguments:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

Quando executado com argumentos apropriados, a soma ou o maior número dos números digitados na linha de comando:

```
$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10
```

Se argumentos inválidos são passados, um erro será emitido:

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

As próximas seções apresentarão detalhes deste exemplo.

Criando um parser

O primeiro passo ao utilizar o *argparse* é criar um objeto *ArgumentParser*:

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

O objeto *ArgumentParser* contém toda informação necerrária para análise e interpretação da linha de comando em tipos de dados Python.

Adicionando argumentos

O preenchimento de *ArgumentParser* com informações sobre os argumentos do programa é feito por chamadas ao método *add_argument()*. Geralmente, estas chamadas informam ao *ArgumentParser* como traduzir strings da linha de comando e torná-los em objetos. Esta informação é armazenada e utilizada quando o método *parse_args()* é invocado. Por exemplo:

```
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                     help='an integer for the accumulator')
>>> parser.add_argument('--sum', dest='accumulate', action='store_const',
...                     const=sum, default=max,
...                     help='sum the integers (default: find the max)')
```

Em seguida, a chamada ao método *parse_args()* irá retornar um objeto com dois atributos, `integers` e `accumulate`. O atributo `integers` será uma lista com um ou mais números inteiros, e o atributo `accumulate` será ou a função *sum()*, se `--sum` for especificado na linha de comando, ou a função *max()*, caso contrário.

Análise de argumentos

ArgumentParser analisa os argumentos através do método *parse_args()*. Isso inspecionará a linha de comando, converterá cada argumento no tipo apropriado e, em seguida, chamará a ação apropriada. Na maioria dos casos, isso significa que um objeto *Namespace* simples será construído a partir de atributos analisados a partir da linha de comando:

```
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

Em um script, *parse_args()* será tipicamente chamado sem argumentos, e *ArgumentParser* irá determinar automaticamente os argumentos de linha de comando de *sys.argv*.

16.4.2 Objetos ArgumentParser

class argparse.**ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=argparse.HelpFormatter*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*)

Cria um novo objeto `ArgumentParser`. Todos os parâmetros devem ser passados como argumentos nomeados. Cada parâmetro tem sua própria descrição mais detalhada abaixo, mas em resumo eles são:

- *prog* - O nome do programa (padrão: `sys.argv[0]`)
- *usage* - A string que descreve o uso do programa (padrão: gerado a partir de argumentos adicionados ao analisador sintático)
- *description* - Texto para exibir antes da ajuda dos argumentos (padrão: nenhuma)
- *epilog* - Texto para exibir após da ajuda dos argumentos (padrão: nenhum)
- *parents* - Uma lista de objetos `ArgumentParser` cujos argumentos também devem ser incluídos
- *formatter_class* - Uma classe para personalizar a saída de ajuda
- *prefix_chars* - O conjunto de caracteres que prefixam argumentos opcionais (padrão: "-")
- *fromfile_prefix_chars* - O conjunto de caracteres que prefixam os arquivos dos quais os argumentos adicionais devem ser lidos (padrão: `None`)
- *argument_default* - O valor padrão global para argumentos (padrão: `None`)
- *conflict_handler* - A estratégia para resolver opcionais conflitantes (geralmente desnecessário)
- *add_help* - Adiciona uma opção `-h/--help` para o analisador sintático (padrão: `True`)
- *allow_abbrev* - Permite que opções longas sejam abreviadas se a abreviação não for ambígua. (padrão: `True`)

Alterado na versão 3.5: O parâmetro *allow_abbrev* foi adicionado.

As seções a seguir descrevem como cada um deles é usado.

prog

Por padrão, os objetos `ArgumentParser` usam `sys.argv[0]` para determinar como exibir o nome do programa nas mensagens de ajuda. Esse padrão é quase sempre desejável porque fará com que as mensagens de ajuda correspondam à forma como o programa foi chamado na linha de comando. Por exemplo, considere um arquivo denominado `myprogram.py` com o seguinte código:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

A ajuda para este programa exibirá `myprogram.py` como o nome do programa (independentemente de onde o programa foi chamado):

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
$ cd ..
```

(continua na próxima página)

(continuação da página anterior)

```
$ python subdir/myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

Para alterar este comportamento padrão, outro valor pode ser fornecido usando o argumento `prog=` para *ArgumentParser*:

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

optional arguments:
 -h, --help  show this help message and exit
```

Observe que o nome do programa, seja determinado a partir de `sys.argv[0]` ou do argumento `prog=`, está disponível para mensagens de ajuda usando o especificador de formato `%(prog)s`.

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo of the myprogram program
```

usage

Por padrão, *ArgumentParser* calcula a mensagem de uso a partir dos argumentos que contém:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
 bar          bar help

optional arguments:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

A mensagem padrão pode ser substituído com o argumento nomeado `usage=`:

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
```

(continua na próxima página)

(continuação da página anterior)

```
 bar           bar help

optional arguments:
 -h, --help    show this help message and exit
 --foo [FOO]   foo help
```

O especificador de formato `%(prog)s` está disponível para preencher o nome do programa em suas mensagens de uso.

description

A maioria das chamadas para o construtor *`ArgumentParser`* usará o argumento nomeado `description=`. Este argumento fornece uma breve descrição do que o programa faz e como funciona. Nas mensagens de ajuda, a descrição é exibida entre a string de uso da linha de comando e as mensagens de ajuda para os vários argumentos:

```
>>> parser = argparse.ArgumentParser(description='A foo that bars')
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
 -h, --help  show this help message and exit
```

Por padrão, a descrição terá sua linha quebrada para que se encaixe no espaço fornecido. Para alterar esse comportamento, consulte o argumento *formatter_class*.

epilog

Alguns programas gostam de exibir uma descrição adicional do programa após a descrição dos argumentos. Esse texto pode ser especificado usando o argumento `epilog=` para *`ArgumentParser`*:

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
 -h, --help  show this help message and exit

And that's how you'd foo a bar
```

Tal como acontece com o argumento *description*, o texto de `epilog=` tem sua quebra de linha habilitada por padrão, mas este comportamento pode ser ajustado com o argumento *formatter_class* para *`ArgumentParser`*.

parents

Às vezes, vários analisadores sintáticos compartilham um conjunto comum de argumentos. Ao invés de repetir as definições desses argumentos, um único analisador com todos os argumentos compartilhados e passado para o argumento `parents=` para *ArgumentParser* pode ser usado. O argumento `parents=` pega uma lista de objetos *ArgumentParser*, coleta todas as ações posicionais e opcionais deles, e adiciona essas ações ao objeto *ArgumentParser* sendo construído:

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

Observe que a maioria dos analisadores sintáticos pais especificará `add_help=False`. Caso contrário, o *ArgumentParser* verá duas opções `-h/--help` (uma no pai e outra no filho) e levantará um erro.

Nota: Você deve inicializar totalmente os analisadores sintáticos antes de passá-los via `parents=`. Se você alterar os analisadores pais após o analisador filho, essas mudanças não serão refletidas no filho.

formatter_class

Objetos *ArgumentParser* permitem que a formação do texto de ajuda seja personalizada por meio da especificação de uma classe de formatação alternativa. Atualmente, há quatro dessas classes:

class argparse.**RawDescriptionHelpFormatter**
class argparse.**RawTextHelpFormatter**
class argparse.**ArgumentDefaultsHelpFormatter**
class argparse.**MetavarTypeHelpFormatter**

RawDescriptionHelpFormatter e *RawTextHelpFormatter* dão mais controle sobre como as descrições textuais são exibidas. Por padrão, objetos *ArgumentParser* quebram em linha os textos *description* e *epilog* nas mensagens de ajuda da linha de comando:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...         was indented weird
...             but that is okay''',
...     epilog='''
...             likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay
```

(continua na próxima página)

(continuação da página anterior)

```
optional arguments:
 -h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

Passar *RawDescriptionHelpFormatter* como `formatter_class=` indica que *description* e *epilog* já estão formatados corretamente e não devem ter suas linhas quebradas:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
...         Please do not mess up this text!
...         --------------------------------
...             I have indented it
...             exactly the way
...             I want it
...         '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
--------------------------------
   I have indented it
   exactly the way
   I want it

optional arguments:
 -h, --help  show this help message and exit
```

RawTextHelpFormatter mantém espaços em branco para todos os tipos de texto de ajuda, incluindo descrições de argumentos. No entanto, várias novas linhas são substituídas por uma. Se você deseja preservar várias linhas em branco, adicione espaços entre as novas linhas.

ArgumentDefaultsHelpFormatter adiciona automaticamente informações sobre os valores padrão para cada uma das mensagens de ajuda do argumento:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar [bar ...]]

positional arguments:
 bar         BAR! (default: [1, 2, 3])

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   FOO! (default: 42)
```

MetavarTypeHelpFormatter usa o nome de argumento *type* para cada argumento como o nome de exibição para seus valores (em vez de usar o *dest* como o formatador regular faz):

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter)
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
usage: PROG [-h] [--foo int] float

positional arguments:
  float

optional arguments:
  -h, --help  show this help message and exit
  --foo int
```

prefix_chars

A maioria das opções de linha de comando usará - como prefixo, por exemplo, `-f/--foo`. Analisadores sintáticos que precisam ter suporte a caracteres de prefixo diferentes ou adicionais, por exemplo, para opções como `+f` ou `/foo`, podem especificá-las usando o argumento `prefix_chars=` para o construtor ArgumentParser:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

O argumento `prefix_chars=` é padronizado como `'-'`. Fornecer um conjunto de caracteres que não inclua - fará com que as opções `-f/--foo` não sejam permitidas.

fromfile_prefix_chars

Sometimes, for example when dealing with a particularly long argument lists, it may make sense to keep the list of arguments in a file rather than typing it out at the command line. If the `fromfile_prefix_chars=` argument is given to the *ArgumentParser* constructor, then arguments that start with any of the specified characters will be treated as files, and will be replaced by the arguments they contain. For example:

```
>>> with open('args.txt', 'w') as fp:
...     fp.write('-f\nbar')
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

Os argumentos lidos de um arquivo devem, por padrão, ser um por linha (mas veja também *convert_arg_line_to_args()*) e são tratados como se estivessem no mesmo lugar que o argumento de referência do arquivo original na linha de comando. Portanto, no exemplo acima, a expressão `['-f', 'foo', '@args.txt']` é considerada equivalente à expressão `['-f', 'foo', '-f', 'bar']`.

O argumento `fromfile_prefix_chars=` é padronizado como `None`, significando que os argumentos nunca serão tratados como referências de arquivo.

argument_default

Geralmente, os padrões dos argumentos são especificados passando um padrão para *add_argument()* ou chamando os métodos *set_defaults()* com um conjunto específico de pares nome-valor. Às vezes, no entanto, pode ser útil especificar um único padrão para todo o analisador para argumentos. Isso pode ser feito passando o argumento nomeado `argument_default=` para *ArgumentParser*. Por exemplo, para suprimir globalmente a criação de atributos em chamadas *parse_args()*, fornecemos `argument_default=SUPPRESS`:

```
>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

allow_abbrev

Normalmente, quando você passa uma lista de argumentos para o método *parse_args()* de um *ArgumentParser*, ele *reconhece as abreviações* de opções longas.

Este recurso pode ser desabilitado configurando `allow_abbrev` para `False`:

```
>>> parser = argparse.ArgumentParser(prog='PROG', allow_abbrev=False)
>>> parser.add_argument('--foobar', action='store_true')
>>> parser.add_argument('--foonley', action='store_false')
>>> parser.parse_args(['--foon'])
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

Novo na versão 3.5.

conflict_handler

Objetos *ArgumentParser* não permitem duas ações com a mesma string de opções. Por padrão, objetos *ArgumentParser* levantam uma exceção se for feita uma tentativa de criar um argumento com uma string de opção que já esteja em uso:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
 ..
ArgumentError: argument --foo: conflicting option string(s): --foo
```

Às vezes (por exemplo, ao usar os *parents*) pode ser útil simplesmente substituir quaisquer argumentos mais antigos com a mesma string de opções. Para obter este comportamento, o valor `'resolve'` pode ser fornecido ao argumento `conflict_handler=` de *ArgumentParser*:

```
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]
```

(continua na próxima página)

(continuação da página anterior)

```
optional arguments:
 -h, --help  show this help message and exit
 -f FOO      old foo help
 --foo FOO   new foo help
```

Observe que os objetos *ArgumentParser* só removem uma ação se todas as suas strings de opção forem substituídas. Assim, no exemplo acima, a antiga ação `-f/--foo` é mantida como a ação `-f`, porque apenas a string de opção `--foo` foi substituída.

add_help

Por padrão, os objetos ArgumentParser adicionam uma opção que simplesmente exibe a mensagem de ajuda do analisador. Por exemplo, considere um arquivo chamado `myprogram.py` contendo o seguinte código:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

Se `-h` ou `--help` for fornecido na linha de comando, a ajuda do ArgumentParser será impressa:

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

Às vezes, pode ser útil desabilitar o acréscimo desta opção de ajuda. Isto pode ser feito passando `False` como o argumento `add_help=` para a classe *ArgumentParser*:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]

optional arguments:
 --foo FOO  foo help
```

A opção de ajuda é normalmente `-h/--help`. A exceção a isso é se o `prefix_chars=` for especificado e não incluir `-`, neste caso `-h` e `--help` não são opções válidas. Neste caso, o primeiro caractere em `prefix_chars` é usado para prefixar as opções de ajuda:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

optional arguments:
  +h, ++help  show this help message and exit
```

16.4.3 O método add_argument()

ArgumentParser.**add_argument**(*name or flags...*[, *action*][, *nargs*][, *const*][, *default*][, *type*][, *choices*][, *required*][, *help*][, *metavar*][, *dest*])

Define como um único argumento de linha de comando deve ser analisado. Cada parâmetro tem sua própria descrição mais detalhada abaixo, mas resumidamente são eles:

- *name or flags* - Um nome ou uma lista de strings de opções, por exemplo. `foo` ou `-f, --foo`.
- *action* - O tipo básico de ação a ser executada quando esse argumento é encontrado na linha de comando.
- *nargs* - O número de argumentos de linha de comando que devem ser consumidos.
- *const* - Um valor constante exigido por algumas seleções *action* e *nargs*.
- *default* - The value produced if the argument is absent from the command line.
- *type* - O tipo para o qual o argumento de linha de comando deve ser convertido.
- *choices* - Um contêiner dos valores permitidos para o argumento.
- *required* - Se a opção de linha de comando pode ou não ser omitida (somente opcionais).
- *help* - Uma breve descrição do que o argumento faz.
- *metavar* - Um nome para o argumento nas mensagens de uso.
- *dest* - O nome do atributo a ser adicionado ao objeto retornado por *parse_args()*.

As seções a seguir descrevem como cada um deles é usado.

name ou flags

O método *add_argument()* define quando um argumento opcional, como `-f` ou `--foo`, ou um argumento posicional, como uma lista de nomes de arquivos, é esperada. Os primeiros argumentos passados ao método *add_argument()* devem ser uma série de flags, ou um simples nome de argumento. Por exemplo, um argumento opcional deve ser criado como:

```
>>> parser.add_argument('-f', '--foo')
```

enquanto um argumento posicional pode ser criado como:

```
>>> parser.add_argument('bar')
```

Quando *parse_args()* é chamado, argumentos opcionais serão identificados pelo prefixo -, e os argumentos restantes serão considerados posicionais:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: the following arguments are required: bar
```

action

Objetos *ArgumentParser* associam argumentos de linha de comando com ações. Essas ações podem fazer praticamente qualquer coisa com os argumentos de linha de comando associados a elas, embora a maioria das ações simplesmente adicione um atributo ao objeto retornado por *parse_args()*. O argumento nomeado `action` especifica como os argumentos da linha de comando devem ser tratados. As ações fornecidas são:

- `'store'` - Isso apenas armazena o valor do argumento. Esta é a ação padrão. Por exemplo:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args('--foo 1'.split())
Namespace(foo='1')
```

- `'store_const'` - Isso armazena o valor especificado pelo argumento nomeado *const*. A ação `'store_const'` é mais comumente usada com argumentos opcionais que especificam algum tipo de sinalizador. Por exemplo:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_const', const=42)
>>> parser.parse_args(['--foo'])
Namespace(foo=42)
```

- `'store_true'` e `'store_false'` - Estes são casos especiais de `'store_const'` usados para armazenar os valores `True` e `False` respectivamente. Além disso, eles criam valores padrão de `False` e `True` respectivamente. Por exemplo:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('--bar', action='store_false')
>>> parser.add_argument('--baz', action='store_false')
>>> parser.parse_args('--foo --bar'.split())
Namespace(foo=True, bar=False, baz=True)
```

- `'append'` - Isso armazena uma lista e anexa cada valor de argumento à lista. Isso é útil para permitir que uma opção seja especificada várias vezes. Exemplo de uso:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

- `'append_const'` - Isso armazena uma lista e anexa o valor especificado pelo argumento nomeado *const* à lista. (Observe que o argumento nomeado *const* tem como padrão `None`.) A ação `'append_const'` é normalmente útil quando vários argumentos precisam armazenar constantes na mesma lista. Por exemplo:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const', const=str)
>>> parser.add_argument('--int', dest='types', action='append_const', const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<class 'str'>, <class 'int'>])
```

- `'count'` - Isso conta o número de vezes que um argumento nomeado ocorre. Por exemplo, isso é útil para aumentar os níveis de verbosidade:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count', default=0)
```

(continua na próxima página)

(continuação da página anterior)

```
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

Observe que o *padrão* será `None`, a menos que seja explicitamente definido como *0*.

- `'help'` - Isso imprime uma mensagem de ajuda completa para todas as opções no analisador sintático atual e sai. Por padrão, uma ação de ajuda é adicionada automaticamente ao analisador sintático. Veja *`ArgumentParser`* para detalhes de como a saída é criada.
- `'version'` - Isso espera um argumento nomeado `version=` na chamada *`add_argument()`* e imprime informações de versão e sai quando invocado:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='%(prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

You may also specify an arbitrary action by passing an Action subclass or other object that implements the same interface. The recommended way to do this is to extend *`Action`*, overriding the `__call__` method and optionally the `__init__` method.

Um exemplo de uma ação personalizada:

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super(FooAction, self).__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
...         print('%r %r %r' % (namespace, values, option_string))
...         setattr(namespace, self.dest, values)
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

Para mais detalhes, veja *`Action`*.

nargs

Os objetos ArgumentParser geralmente associam um único argumento de linha de comando a uma única ação a ser executada. O argumento nomeado `nargs` associa um número diferente de argumentos de linha de comando com uma única ação. Os valores suportados são:

- `N` (um inteiro). Os argumentos `N` da linha de comando serão reunidos em uma lista. Por exemplo:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs=2)
>>> parser.add_argument('bar', nargs=1)
>>> parser.parse_args('c --foo a b'.split())
Namespace(bar=['c'], foo=['a', 'b'])
```

Observe que `nargs=1` produz uma lista de um item. Isso é diferente do padrão, em que o item é produzido sozinho.

- `'?'`. Um argumento será consumido da linha de comando, se possível, e produzido como um único item. Se nenhum argumento de linha de comando estiver presente, o valor de *default* será produzido. Observe que, para argumentos opcionais, há um caso adicional - a string de opções está presente, mas não é seguida por um argumento de linha de comando. Neste caso o valor de *const* será produzido. Alguns exemplos para ilustrar isso:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='?', const='c', default='d')
>>> parser.add_argument('bar', nargs='?', default='d')
>>> parser.parse_args(['XX', '--foo', 'YY'])
Namespace(bar='XX', foo='YY')
>>> parser.parse_args(['XX', '--foo'])
Namespace(bar='XX', foo='c')
>>> parser.parse_args([])
Namespace(bar='d', foo='d')
```

Um dos usos mais comuns de `nargs='?'` é permitir arquivos de entrada e saída opcionais:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', nargs='?', type=argparse.FileType('r'),
...                     default=sys.stdin)
>>> parser.add_argument('outfile', nargs='?', type=argparse.FileType('w'),
...                     default=sys.stdout)
>>> parser.parse_args(['input.txt', 'output.txt'])
Namespace(infile=<_io.TextIOWrapper name='input.txt' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='output.txt' encoding='UTF-8'>)
>>> parser.parse_args([])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='<stdout>' encoding='UTF-8'>)
```

- `'*'`. Todos os argumentos de linha de comando presentes são reunidos em uma lista. Note que geralmente não faz muito sentido ter mais de um argumento posicional com `nargs='*'`, mas vários argumentos opcionais com `nargs='*'` são possíveis. Por exemplo:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

- `'+'`. Assim como `'*'`, todos os argumentos de linha de comando presentes são reunidos em uma lista. Além disso, uma mensagem de erro será gerada se não houver pelo menos um argumento de linha de comando presente. Por exemplo:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: the following arguments are required: foo
```

- `argparse.REMAINDER`. All the remaining command-line arguments are gathered into a list. This is commonly useful for command line utilities that dispatch to other command line utilities:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo')
>>> parser.add_argument('command')
>>> parser.add_argument('args', nargs=argparse.REMAINDER)
>>> print(parser.parse_args('--foo B cmd --arg1 XX ZZ'.split()))
Namespace(args=['--arg1', 'XX', 'ZZ'], command='cmd', foo='B')
```

Se o argumento nomeado `nargs` não for fornecido, o número de argumentos consumidos é determinado pela *action*. Geralmente, isso significa que um único argumento de linha de comando será consumido e um único item (não uma lista) será produzido.

const

O argumento `const` de *`add_argument()`* é usado para manter valores constantes que não são lidos da linha de comando, mas são necessários para as várias ações *`ArgumentParser`*. Os dois usos mais comuns são:

- Quando *`add_argument()`* é chamado com `action='store_const'` ou `action='append_const'`. Essas ações adicionam o valor `const` a um dos atributos do objeto retornado por *`parse_args()`*. Consulte a descrição da *action* para obter exemplos.
- Quando *`add_argument()`* é chamado com strings de opções (como `-f` ou `--foo`) e `nargs='?'`. Isso cria um argumento opcional que pode ser seguido por zero ou um argumento de linha de comando. Ao analisar a linha de comando, se a string de opções for encontrada sem nenhum argumento de linha de comando seguindo, o valor de `const` será assumido. Veja a descrição de *nargs* para exemplos.

Com as ações `'store_const'` e `'append_const'`, o argumento nomeado `const` deve ser fornecido. Para outras ações, o padrão é `None`.

default

Todos os argumentos opcionais e alguns argumentos posicionais podem ser omitidos na linha de comando. O argumento nomeado `default` de *`add_argument()`*, cujo valor padrão é `None`, especifica qual valor deve ser usado se o argumento de linha de comando não estiver presente. Para argumentos opcionais, o valor `default` é usado quando a string de opção não estava presente na linha de comando:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

Se o valor `default` for uma string, o analisador analisa o valor como se fosse um argumento de linha de comando. Em particular, o analisador aplica qualquer argumento de conversão *type*, se fornecido, antes de definir o atributo no valor de retorno *`Namespace`*. Caso contrário, o analisador usa o valor como está:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

For positional arguments with *nargs* equal to `?` or `*`, the `default` value is used when no command-line argument was present:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

Providing `default=argparse.SUPPRESS` causes no attribute to be added if the command-line argument was not present:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

tipo

By default, *ArgumentParser* objects read command-line arguments in as simple strings. However, quite often the command-line string should instead be interpreted as another type, like a *float* or *int*. The `type` keyword argument of *add_argument()* allows any necessary type-checking and type conversions to be performed. Common built-in types and functions can be used directly as the value of the `type` argument:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.add_argument('bar', type=open)
>>> parser.parse_args('2 temp.txt'.split())
Namespace(bar=<_io.TextIOWrapper name='temp.txt' encoding='UTF-8'>, foo=2)
```

See the section on the *default* keyword argument for information on when the `type` argument is applied to default arguments.

To ease the use of various types of files, the argparse module provides the factory FileType which takes the `mode=`, `bufsize=`, `encoding=` and `errors=` arguments of the *open()* function. For example, `FileType('w')` can be used to create a writable file:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar', type=argparse.FileType('w'))
>>> parser.parse_args(['out.txt'])
Namespace(bar=<_io.TextIOWrapper name='out.txt' encoding='UTF-8'>)
```

`type=` can take any callable that takes a single string argument and returns the converted value:

```
>>> def perfect_square(string):
...     value = int(string)
...     sqrt = math.sqrt(value)
...     if sqrt != int(sqrt):
...         msg = "%r is not a perfect square" % string
...         raise argparse.ArgumentTypeError(msg)
...     return value
...
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=perfect_square)
>>> parser.parse_args(['9'])
```

(continua na próxima página)

(continuação da página anterior)

```
Namespace(foo=9)
>>> parser.parse_args(['7'])
usage: PROG [-h] foo
PROG: error: argument foo: '7' is not a perfect square
```

The *choices* keyword argument may be more convenient for type checkers that simply check against a range of values:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=int, choices=range(5, 10))
>>> parser.parse_args(['7'])
Namespace(foo=7)
>>> parser.parse_args(['11'])
usage: PROG [-h] {5,6,7,8,9}
PROG: error: argument foo: invalid choice: 11 (choose from 5, 6, 7, 8, 9)
```

See the *choices* section for more details.

choices

Some command-line arguments should be selected from a restricted set of values. These can be handled by passing a container object as the *choices* keyword argument to *add_argument()*. When the command line is parsed, argument values will be checked, and an error message will be displayed if the argument was not one of the acceptable values:

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')
```

Note that inclusion in the *choices* container is checked after any *type* conversions have been performed, so the type of the objects in the *choices* container should match the *type* specified:

```
>>> parser = argparse.ArgumentParser(prog='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
>>> parser.parse_args(['4'])
usage: doors.py [-h] {1,2,3}
doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)
```

Any object that supports the `in` operator can be passed as the *choices* value, so *dict* objects, *set* objects, custom containers, etc. are all supported.

required

In general, the *argparse* module assumes that flags like `-f` and `--bar` indicate *optional* arguments, which can always be omitted at the command line. To make an option *required*, `True` can be specified for the `required=` keyword argument to *add_argument()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: argparse.py [-h] [--foo FOO]
argparse.py: error: option --foo is required
```

As the example shows, if an option is marked as `required`, *parse_args()* will report an error if that option is not present at the command line.

Nota: Required options are generally considered bad form because users expect *options* to be *optional*, and thus they should be avoided when possible.

---

help

The `help` value is a string containing a brief description of the argument. When a user requests help (usually by using `-h` or `--help` at the command line), these `help` descriptions will be displayed with each argument:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', action='store_true',
...                     help='foo the bars before frobbling')
>>> parser.add_argument('bar', nargs='+',
...                     help='one of the bars to be frobbled')
>>> parser.parse_args(['-h'])
usage: frobble [-h] [--foo] bar [bar ...]

positional arguments:
 bar     one of the bars to be frobbled

optional arguments:
 -h, --help  show this help message and exit
 --foo   foo the bars before frobbling
```

The `help` strings can include various format specifiers to avoid repetition of things like the program name or the argument *default*. The available specifiers include the program name, `%(prog)s` and most keyword arguments to *add_argument()*, e.g. `%(default)s`, `%(type)s`, etc.:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                     help='the bar to %(prog)s (default: %(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
 bar     the bar to frobble (default: 42)

optional arguments:
 -h, --help  show this help message and exit
```

As the help string supports %-formatting, if you want a literal `%` to appear in the help string, you must escape it as `%%`.

argparse supports silencing the help entry for certain options, by setting the `help` value to `argparse.SUPPRESS`:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

optional arguments:
  -h, --help  show this help message and exit
```

metavar

When *ArgumentParser* generates help messages, it needs some way to refer to each expected argument. By default, ArgumentParser objects use the *dest* value as the "name" of each object. By default, for positional argument actions, the *dest* value is used directly, and for optional argument actions, the *dest* value is uppercased. So, a single positional argument with `dest='bar'` will be referred to as `bar`. A single optional argument `--foo` that should be followed by a single command-line argument will be referred to as `FOO`. An example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo FOO] bar

positional arguments:
 bar

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO
```

An alternative name can be specified with `metavar`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo YYY] XXX

positional arguments:
 XXX

optional arguments:
 -h, --help  show this help message and exit
 --foo YYY
```

Note that `metavar` only changes the *displayed* name - the name of the attribute on the *parse_args()* object is still determined by the *dest* value.

Different values of `nargs` may cause the metavar to be used multiple times. Providing a tuple to `metavar` specifies a different display for each of the arguments:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
>>> parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]

optional arguments:
 -h, --help     show this help message and exit
 -x X X
 --foo bar baz
```

dest

Most *ArgumentParser* actions add some value as an attribute of the object returned by *parse_args()*. The name of this attribute is determined by the `dest` keyword argument of *add_argument()*. For positional argument actions, `dest` is normally supplied as the first argument to *add_argument()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

For optional argument actions, the value of `dest` is normally inferred from the option strings. *ArgumentParser* generates the value of `dest` by taking the first long option string and stripping away the initial `--` string. If no long option strings were supplied, `dest` will be derived from the first short option string by stripping the initial `-` character. Any internal `-` characters will be converted to `_` characters to make sure the string is a valid attribute name. The examples below illustrate this behavior:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

`dest` allows a custom attribute name to be provided:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

Action classes

Action classes implement the Action API, a callable which returns a callable which processes arguments from the command-line. Any object which follows this API may be passed as the `action` parameter to `add_argument()`.

class argparse.**Action**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Action objects are used by an ArgumentParser to represent the information needed to parse a single argument from one or more strings from the command line. The Action class must accept the two positional arguments plus any keyword arguments passed to *ArgumentParser.add_argument()* except for the `action` itself.

Instances of Action (or return value of any callable to the `action` parameter) should have attributes "dest", "option_strings", "default", "type", "required", "help", etc. defined. The easiest way to ensure these attributes are defined is to call `Action.__init__`.

Action instances should be callable, so subclasses must override the `__call__` method, which should accept four parameters:

- `parser` - The ArgumentParser object which contains this action.
- `namespace` - The *`Namespace`* object that will be returned by *`parse_args()`*. Most actions add an attribute to this object using *`setattr()`*.
- `values` - The associated command-line arguments, with any type conversions applied. Type conversions are specified with the *type* keyword argument to *`add_argument()`*.
- `option_string` - The option string that was used to invoke this action. The `option_string` argument is optional, and will be absent if the action is associated with a positional argument.

The `__call__` method may perform arbitrary actions, but will typically set attributes on the `namespace` based on `dest` and `values`.

16.4.4 The parse_args() method

`ArgumentParser.`**`parse_args`**(*args=None*, *namespace=None*)

Convert argument strings to objects and assign them as attributes of the namespace. Return the populated namespace.

Previous calls to *`add_argument()`* determine exactly what objects are created and how they are assigned. See the documentation for *`add_argument()`* for details.

- *args* - List of strings to parse. The default is taken from *`sys.argv`*.
- *namespace* - An object to take the attributes. The default is a new empty *`Namespace`* object.

Option value syntax

The *`parse_args()`* method supports several ways of specifying the value of an option (if it takes one). In the simplest case, the option and its value are passed as two separate arguments:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

For long options (options with names longer than a single character), the option and value can also be passed as a single command-line argument, using = to separate them:

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

For short options (options only one character long), the option and its value can be concatenated:

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

Several short options can be joined together, using only a single - prefix, as long as only the last option (or none of them) requires a value:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
>>> parser.add_argument('-z')
>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

Argumentos inválidos

While parsing the command line, *parse_args()* checks for a variety of errors, including ambiguous options, invalid types, invalid options, wrong number of positional arguments, etc. When it encounters such an error, it exits and prints the error along with a usage message:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

>>> # invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

Argumentos contendo -

The *parse_args()* method attempts to give errors whenever the user has clearly made a mistake, but some situations are inherently ambiguous. For example, the command-line argument `-1` could either be an attempt to specify an option or an attempt to provide a positional argument. The *parse_args()* method is cautious here: positional arguments may only begin with - if they look like negative numbers and there are no options in the parser that look like negative numbers:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')
```

(continua na próxima página)

(continuação da página anterior)

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

If you have positional arguments that must begin with `-` and don't look like negative numbers, you can insert the pseudo-argument `'--'` which tells *parse_args()* that everything after that is a positional argument:

```
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

Argument abbreviations (prefix matching)

The *parse_args()* method *by default* allows long options to be abbreviated to a prefix, if the abbreviation is unambiguous (the prefix matches a unique option):

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

An error is produced for arguments that could produce more than one options. This feature can be disabled by setting *allow_abbrev* to `False`.

Além do `sys.argv`

Sometimes it may be useful to have an ArgumentParser parse arguments other than those of *sys.argv*. This can be accomplished by passing a list of strings to *parse_args()*. This is useful for testing at the interactive prompt:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=range(10),
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const', const=sum,
...     default=max, help='sum the integers (default: find the max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

O objeto Namespace

class argparse.**Namespace**

Simple class used by default by *parse_args()* to create an object holding attributes and return it.

This class is deliberately simple, just an *object* subclass with a readable string representation. If you prefer to have dict-like view of the attributes, you can use the standard Python idiom, *vars()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

It may also be useful to have an *ArgumentParser* assign attributes to an already existing object, rather than a new *Namespace* object. This can be achieved by specifying the `namespace=` keyword argument:

```
>>> class C:
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

16.4.5 Other utilities

Sub-comandos

ArgumentParser.**add_subparsers**([*title*][, *description*][, *prog*][, *parser_class*][, *action*][, *option_string*][, *dest*][, *required*][, *help*][, *metavar*])

Many programs split up their functionality into a number of sub-commands, for example, the `svn` program can invoke sub-commands like `svn checkout`, `svn update`, and `svn commit`. Splitting up functionality this way can be a particularly good idea when a program performs several different functions which require different kinds of command-line arguments. *ArgumentParser* supports the creation of such sub-commands with

the *add_subparsers()* method. The *add_subparsers()* method is normally called with no arguments and returns a special action object. This object has a single method, `add_parser()`, which takes a command name and any *ArgumentParser* constructor arguments, and returns an *ArgumentParser* object that can be modified as usual.

Descrição de parâmetros:

- title - title for the sub-parser group in help output; by default "subcommands" if description is provided, otherwise uses title for positional arguments
- description - description for the sub-parser group in help output, by default `None`
- prog - usage information that will be displayed with sub-command help, by default the name of the program and any positional arguments before the subparser argument
- parser_class - class which will be used to create sub-parser instances, by default the class of the current parser (e.g. ArgumentParser)
- *action* - the basic type of action to be taken when this argument is encountered at the command line
- *dest* - name of the attribute under which sub-command name will be stored; by default `None` and no value is stored
- *required* - Whether or not a subcommand must be provided, by default `False` (added in 3.7)
- *help* - help for sub-parser group in help output, by default `None`
- *metavar* - string presenting available sub-commands in help; by default it is `None` and presents sub-commands in form {cmd1, cmd2, ..}

Alguns exemplos de uso:

```
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='sub-command help')
>>>
>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices='XYZ', help='baz help')
>>>
>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

Note that the object returned by *parse_args()* will only contain attributes for the main parser and the subparser that was selected by the command line (and not any other subparsers). So in the example above, when the `a` command is specified, only the `foo` and `bar` attributes are present, and when the `b` command is specified, only the `foo` and `baz` attributes are present.

Similarly, when a help message is requested from a subparser, only the help for that particular parser will be printed. The help message will not include parent parser or sibling parser messages. (A help message for each subparser command, however, can be given by supplying the `help=` argument to `add_parser()` as above.)

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}   sub-command help
    a     a help
    b     b help

optional arguments:
  -h, --help  show this help message and exit
  --foo   foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar     bar help

optional arguments:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

optional arguments:
  -h, --help     show this help message and exit
  --baz {X,Y,Z}  baz help
```

The *add_subparsers()* method also supports `title` and `description` keyword arguments. When either is present, the subparser's commands will appear in their own group in the help output. For example:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                    description='valid subcommands',
...                                    help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

optional arguments:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}   additional help
```

Furthermore, `add_parser` supports an additional `aliases` argument, which allows multiple strings to refer to the same subparser. This example, like `svn`, aliases `co` as a shorthand for `checkout`:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>> checkout = subparsers.add_parser('checkout', aliases=['co'])
>>> checkout.add_argument('foo')
>>> parser.parse_args(['co', 'bar'])
Namespace(foo='bar')
```

One particularly effective way of handling sub-commands is to combine the use of the *add_subparsers()* method with calls to *set_defaults()* so that each subparser knows which Python function it should execute. For example:

```
>>> # sub-command functions
>>> def foo(args):
...     print(args.x * args.y)
...
>>> def bar(args):
...     print('((%s))' % args.z)
...
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>>
>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
2.0
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

This way, you can let *parse_args()* do the job of calling the appropriate function after argument parsing is complete. Associating functions with actions like this is typically the easiest way to handle the different actions for each of your subparsers. However, if it is necessary to check the name of the subparser that was invoked, the `dest` keyword argument to the *add_subparsers()* call will work:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

Alterado na versão 3.7: New *required* keyword argument.

Objetos FileType

class argparse.**FileType**(*mode='r'*, *bufsize=-1*, *encoding=None*, *errors=None*)

The *FileType* factory creates objects that can be passed to the type argument of *ArgumentParser.add_argument()*. Arguments that have *FileType* objects as their type will open command-line arguments as files with the requested modes, buffer sizes, encodings and error handling (see the *open()* function for more details):

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--raw', type=argparse.FileType('wb', 0))
>>> parser.add_argument('out', type=argparse.FileType('w', encoding='UTF-8'))
>>> parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8'>, raw=
↪<_io.FileIO name='raw.dat' mode='wb'>)
```

FileType objects understand the pseudo-argument `'-'` and automatically convert this into `sys.stdin` for readable *FileType* objects and `sys.stdout` for writable *FileType* objects:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>)
```

Novo na versão 3.4: The *encodings* and *errors* keyword arguments.

Grupos de Argumentos

ArgumentParser.**add_argument_group**(*title=None*, *description=None*)

By default, *ArgumentParser* groups command-line arguments into "positional arguments" and "optional arguments" when displaying help messages. When there is a better conceptual grouping of arguments than this default one, appropriate groups can be created using the *add_argument_group()* method:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
  bar    bar help
  --foo FOO  foo help
```

The *add_argument_group()* method returns an argument group object which has an *add_argument()* method just like a regular *ArgumentParser*. When an argument is added to the group, the parser treats it just like a normal argument, but displays the argument in a separate group for help messages. The *add_argument_group()* method accepts *title* and *description* arguments which can be used to customize this display:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
```

(continua na próxima página)

(continuação da página anterior)

```
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo    foo help

group2:
  group2 description

  --bar BAR  bar help
```

Note that any arguments not in your user-defined groups will end up back in the usual "positional arguments" and "optional arguments" sections.

Exclusão Mútua

`ArgumentParser.`**`add_mutually_exclusive_group`**(*required=False*)

Create a mutually exclusive group. *argparse* will make sure that only one of the arguments in the mutually exclusive group was present on the command line:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

The *add_mutually_exclusive_group()* method also accepts a *required* argument, to indicate that at least one of the mutually exclusive arguments is required:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

Note that currently mutually exclusive argument groups do not support the *title* and *description* arguments of *add_argument_group()*.

Parser defaults

ArgumentParser.**set_defaults**(***kwargs*)

Most of the time, the attributes of the object returned by *parse_args()* will be fully determined by inspecting the command-line arguments and the argument actions. *set_defaults()* allows some additional attributes that are determined without any inspection of the command line to be added:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

Note that parser-level defaults always override argument-level defaults:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

Parser-level defaults can be particularly useful when working with multiple parsers. See the *add_subparsers()* method for an example of this type.

ArgumentParser.**get_default**(*dest*)

Get the default value for a namespace attribute, as set by either *add_argument()* or by *set_defaults()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

Imprimindo a ajuda

In most typical applications, *parse_args()* will take care of formatting and printing any usage or error messages. However, several formatting methods are available:

ArgumentParser.**print_usage**(*file=None*)

Print a brief description of how the *ArgumentParser* should be invoked on the command line. If *file* is `None`, *sys.stdout* is assumed.

ArgumentParser.**print_help**(*file=None*)

Print a help message, including the program usage and information about the arguments registered with the *ArgumentParser*. If *file* is `None`, *sys.stdout* is assumed.

There are also variants of these methods that simply return a string instead of printing it:

ArgumentParser.**format_usage**()

Return a string containing a brief description of how the *ArgumentParser* should be invoked on the command line.

ArgumentParser.**format_help**()

Return a string containing a help message, including the program usage and information about the arguments registered with the *ArgumentParser*.

Análise parcial

ArgumentParser.**parse_known_args**(*args=None*, *namespace=None*)

Sometimes a script may only parse a few of the command-line arguments, passing the remaining arguments on to another script or program. In these cases, the *parse_known_args()* method can be useful. It works much like *parse_args()* except that it does not produce an error when extra arguments are present. Instead, it returns a two item tuple containing the populated namespace and the list of remaining argument strings.

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

Aviso: *Prefix matching* rules apply to `parse_known_args()`. The parser may consume an option even if it's just a prefix of one of its known options, instead of leaving it in the remaining arguments list.

Customizing file parsing

ArgumentParser.**convert_arg_line_to_args**(*arg_line*)

Arguments that are read from a file (see the *fromfile_prefix_chars* keyword argument to the *ArgumentParser* constructor) are read one argument per line. *convert_arg_line_to_args()* can be overridden for fancier reading.

This method takes a single argument *arg_line* which is a string read from the argument file. It returns a list of arguments parsed from this string. The method is called once per line read from the argument file, in order.

A useful override of this method is one that treats each space-separated word as an argument. The following example demonstrates how to do this:

```
class MyArgumentParser(argparse.ArgumentParser):
    def convert_arg_line_to_args(self, arg_line):
        return arg_line.split()
```

Métodos existentes

ArgumentParser.**exit**(*status=0*, *message=None*)

This method terminates the program, exiting with the specified *status* and, if given, it prints a *message* before that.

ArgumentParser.**error**(*message*)

This method prints a usage message including the *message* to the standard error and terminates the program with a status code of 2.

Intermixed parsing

ArgumentParser.**parse_intermixed_args**(*args=None*, *namespace=None*)

ArgumentParser.**parse_known_intermixed_args**(*args=None*, *namespace=None*)

A number of Unix commands allow the user to intermix optional arguments with positional arguments. The *parse_intermixed_args()* and *parse_known_intermixed_args()* methods support this parsing style.

These parsers do not support all the argparse features, and will raise exceptions if unsupported features are used. In particular, subparsers, `argparse.REMAINDER`, and mutually exclusive groups that include both optionals and positionals are not supported.

The following example shows the difference between *parse_known_args()* and *parse_intermixed_args()*: the former returns `['2', '3']` as unparsed arguments, while the latter collects all the positionals into `rest`.

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('cmd')
>>> parser.add_argument('rest', nargs='*', type=int)
>>> parser.parse_known_args('doit 1 --foo bar 2 3'.split())
(Namespace(cmd='doit', foo='bar', rest=[1]), ['2', '3'])
>>> parser.parse_intermixed_args('doit 1 --foo bar 2 3'.split())
Namespace(cmd='doit', foo='bar', rest=[1, 2, 3])
```

parse_known_intermixed_args() returns a two item tuple containing the populated namespace and the list of remaining argument strings. *parse_intermixed_args()* raises an error if there are any remaining unparsed argument strings.

Novo na versão 3.7.

16.4.6 Upgrading optparse code

Originally, the *argparse* module had attempted to maintain compatibility with *optparse*. However, *optparse* was difficult to extend transparently, particularly with the changes required to support the new `nargs=` specifiers and better usage messages. When most everything in *optparse* had either been copy-pasted over or monkey-patched, it no longer seemed practical to try to maintain the backwards compatibility.

The *argparse* module improves on the standard library *optparse* module in a number of ways including:

- Tratando argumentos posicionais.
- Supporting sub-commands.
- Allowing alternative option prefixes like + and `/`.
- Handling zero-or-more and one-or-more style arguments.
- Producing more informative usage messages.
- Providing a much simpler interface for custom `type` and `action`.

A partial upgrade path from *optparse* to *argparse*:

- Replace all *optparse.OptionParser.add_option()* calls with *ArgumentParser.add_argument()* calls.
- Replace `(options, args) = parser.parse_args()` with `args = parser.parse_args()` and add additional *ArgumentParser.add_argument()* calls for the positional arguments. Keep in mind that what was previously called `options`, now in the *argparse* context is called `args`.

- Replace *optparse.OptionParser.disable_interspersed_args()* by using *parse_intermixed_args()* instead of *parse_args()*.
- Replace callback actions and the `callback_*` keyword arguments with `type` or `action` arguments.
- Replace string names for `type` keyword arguments with the corresponding type objects (e.g. int, float, complex, etc).
- Replace `optparse.Values` with *Namespace* and `optparse.OptionError` and `optparse.OptionValueError` with `ArgumentError`.
- Replace strings with implicit arguments such as `%default` or `%prog` with the standard Python syntax to use dictionaries to format strings, that is, `%(default)s` and `%(prog)s`.
- Replace the OptionParser constructor `version` argument with a call to `parser.add_argument('--version', action='version', version='<the version>')`.

16.5 `getopt` — Analisador sintático no estilo C para opções de linha de comando

Código Fonte: Lib/getopt.py

Nota: O módulo *getopt* é um analisador sintático para opções de linha de comando cuja API é projetada para ser familiar aos usuários da função C `getopt()`. Os usuários que não estão familiarizados com a função C `getopt()` ou que gostariam de escrever menos código e obter mensagens de ajuda e de erro melhores devem considerar o uso do módulo *argparse*.

Este módulo ajuda os scripts a analisar os argumentos da linha de comando em `sys.argv`. Ele suporta as mesmas convenções da função Unix `getopt()` (incluindo os significados especiais de argumentos da forma '-' e '--'). Longas opções semelhantes às suportadas pelo software GNU também podem ser usadas por meio de um terceiro argumento opcional.

Este módulo fornece duas funções e uma exceção:

`getopt.`**`getopt`**(*args*, *shortopts*, *longopts=[]*)

Analisa opções de linha de comando e lista de parâmetros. *args* é a lista de argumentos a ser analisada, sem a referência inicial para o programa em execução. Normalmente, isso significa `sys.argv[1:]`. *shortopts* é a string de letras de opção que o script deseja reconhecer, com opções que requerem um argumento seguido por dois pontos (`':'`; ou seja, o mesmo formato que Unix `getopt()` usa).

Nota: Ao contrário do GNU `getopt()`, após um argumento sem opção, todos os argumentos adicionais são considerados também sem opção. Isso é semelhante à maneira como os sistemas Unix não GNU funcionam.

---

longopts, se especificado, deve ser uma lista de strings com os nomes das opções longas que devem ser suportadas. Os caracteres `'--'` no início não devem ser incluídos no nome da opção. Opções longas que requerem um argumento devem ser seguidas por um sinal de igual (`'='`). Argumentos opcionais não são suportados. Para aceitar apenas opções longas, *shortopts* deve ser uma string vazia. Opções longas na linha de comando podem ser reconhecidas, desde que forneçam um prefixo do nome da opção que corresponda exatamente a uma das opções aceitas. Por exemplo, se *longopts* for `['foo', 'frob']`, a opção `--fo` irá corresponder a `--foo`, mas `--f` não corresponderá exclusivamente, então *GetoptError* será levantada.

O valor de retorno consiste em dois elementos: o primeiro é uma lista de pares `(option, value)`; a segunda é a lista de argumentos de programa restantes depois que a lista de opções foi removida (esta é uma fatia ao final de *args*). Cada par de opção e valor retornado tem a opção como seu primeiro elemento, prefixado com um hífen para opções curtas (por exemplo, `'-x'`) ou dois hifenes para opções longas (por exemplo, `'--long-option'`), e o argumento da opção como seu segundo elemento, ou uma string vazia se a opção não tiver argumento. As opções ocorrem na lista na mesma ordem em que foram encontradas, permitindo assim múltiplas ocorrências. Opções longas e curtas podem ser misturadas.

`getopt.`**`gnu_getopt`**(*args*, *shortopts*, *longopts=[]*)

Esta função funciona como *`getopt()`*, exceto que o modo de digitalização do estilo GNU é usado por padrão. Isso significa que os argumentos de opção e não opção podem ser misturados. A função *`getopt()`* interrompe o processamento das opções assim que um argumento não opcional é encontrado.

Se o primeiro caractere da string de opção for `'+'`, ou se a variável de ambiente `POSIXLY_CORRECT` estiver definida, então o processamento da opção para assim que um argumento não opcional for encontrado.

exception `getopt.`**`GetoptError`**

Isso é levantado quando uma opção não reconhecida é encontrada na lista de argumentos ou quando uma opção que requer um argumento não é fornecida. O argumento para a exceção é uma string que indica a causa do erro. Para opções longas, um argumento dado a uma opção que não requer uma também fará com que essa exceção seja levantada. Os atributos `msg` e `opt` fornecem a mensagem de erro e a opção relacionada; se não houver uma opção específica à qual a exceção se relaciona, `opt` é uma string vazia.

exception `getopt.`**`error`**

Alias para *`GetoptError`*; para compatibilidade reversa.

Um exemplo usando apenas opções de estilo Unix:

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

Usar nomes de opções longos é igualmente fácil:

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def'), ('-x', '')]
>>> args
['a1', 'a2']
```

Em um script, o uso típico é algo assim:

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
```

(continua na próxima página)

(continuação da página anterior)

```
    except getopt.GetoptError as err:
        # print help information and exit:
        print(err)  # will print something like "option -a not recognized"
        usage()
        sys.exit(2)
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-o", "--output"):
            output = a
        else:
            assert False, "unhandled option"
    # ...

if __name__ == "__main__":
    main()
```

Observe que uma interface de linha de comando equivalente pode ser produzida com menos código e mais mensagens de erro de ajuda e erro informativas usando o módulo *argparse*:

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    args = parser.parse_args()
    # ... do something with args.output ...
    # ... do something with args.verbose ..
```

Ver também:

Módulo *argparse* Alternativa de opção de linha de comando e biblioteca de análise de argumento.

16.6 `logging` — Facilidade para registrar com Python

Código Fonte: Lib/logging/__init__.py

Important

Esta página contêm informação de referência da API. Para informação tutorial e discussão de tópicos mais avançados, consulte

- Basic Tutorial
- Advanced Tutorial
- Logging Cookbook

Este módulo define funções e classes que implementam um registro flexível de eventos de sistema para aplicações e bibliotecas.

O principal benefício de ter a API de registro de eventos a partir de um módulo da biblioteca padrão é que todos os módulos Python podem participar no registro de eventos, de forma que sua aplicação pode incluir suas próprias mensagens, integradas com mensagens de módulos de terceiros.

O módulo fornece muita funcionalidade e flexibilidade. Se não for familiar com registro de eventos, a melhor forma de lidar com isso é verificar os tutoriais (siga os links à direita).

As classes básicas definidas no módulo, juntamente com suas funções, são listdas abaixo.

- Loggers expõem a interface que o código da aplicação usa diretamente.
- Handlers enviam os registros do evento (criados por loggers) aos destinos apropriados.
- Filters fornecem uma facilidade granular para determinar quais registros de eventos enviar à saída.
- Formatters especificam o layout dos registros de eventos na saída final.

16.6.1 Objetos Logger

Loggers tem os atributos e métodos a seguir. Observem que Loggers *NUNCA* devem ser instanciados diretamete, mas sempre através da função `logging.getLogger(name)`. Múltiplas chamadas à função *`getLogger()`* com o mesmo nome sempre retornará uma referência para o mesmo objeto Logger.

O `name` é, potencialmente, um valor hierárquico separado por ponto, como `foo.bar.baz` (embora também possa ser simplesmente `foo`, por exemplo). Loggers que estão mais abaixo na lista hierárquica são filhos de loggers que estão acima na lista. Por exemplo, dado um logger com nome `foo`, loggers com nomes de `foo.bar`, `foo.bar.baz`, e `foo.bam` são todos descendentes de `foo`. A hierarquia do nome do logger é análoga às dos pacotes Python, e idêntico a esses, se você organizar seus loggers baseado nos módulos usando a construção recomendada `logging.getLogger(__name__)`. Isso porque num módulo `__name__` é o nome do módulo no espaço de nomes do pacote Python.

class logging.**Logger**

propagate
Se este atributo for avaliado como verdadeiro, os registros de eventos para esse logger serão repassados para loggers de níveis superiores (ancestrais), em adição a qualquer destino configurado para esse logger.

Se o valor for falso, as mensagens de registro de eventos não são passadas para loggers ancestrais.

O construtor atribui este valor como `True`.

Nota: Se você configurar um destino para o logger *e* um ou mais dos ancestrais, pode acontecer a mesma mensagem registrada várias vezes. Em geral, você não precisa configurar saídas para mais que um logger - se você configurar apenas para o logger principal da hierarquia, então todos os eventos dos loggers descendentes serão visualizados ali, fornecido pela configuração de propagação, cujo padrão é `True`. Um cenário comum é configurar as saídas somente no logger raiz, e deixar a propagação tomar conta do resto.

---

setLevel(*level*)
Ajuste o limite para este logger para *level*. Mensagens de registro de eventos que forem menos severas que este *level* serão ignoradas; mensagens que tenham nível de severidade igual ou maior que *level* serão emitidas para os destinos de saída configurados para o logger, a menos que o nível da saída tenha sido configurado para uma severidade ainda maior.

Quando um logger é criado, o nível é definido como NOTSET (que faz com que todas as mensagens sejam processadas quando o logger for o logger raiz, ou delegação para o pai quando o logger não for um logger raiz). Observe que o logger raiz é criado com nível WARNING.

O termo 'delegado ao parente' significa que o logger possui um nível de NOTSET, e a sua cadeia de loggers ancestrais será percorrida até que um ancestral com o nível diferente de NOTSET seja encontrado ou até a raiz ser alcançada.

Se um ancestral for achado com um nível diferente de NOTSET, então o nível daquele ancestral será tratado como o nível efetivo do logger que começou a busca por ancestrais, e é usado para determinar como um registro de evento será manipulado.

Se a raiz é alcançada e o seu nível é NOTSET, então todas as mensagens serão processadas. Caso contrário, o nível da raiz será usada como o nível efetivo.

Veja *Logging Levels* para uma lista de níveis.

Alterado na versão 3.2: O parâmetro *level* agora é aceito como uma string representando o nível tal qual 'INFO', como uma alternativa as constantes inteiras tal qual INFO. Note, entretanto, que os níveis são guardados internamente como inteiros e os métodos como, por exemplo, *getEffectiveLevel()* e *isEnabledFor()* retornarão/esperarão que sejam passados inteiros.

isEnabledFor(*level*)
Indica se a mensagem com gravidade *level* seria processada por esse logger. Esse método checa primeiro o nível à nível de módulo definido por logging.disable(level) e então o nível efetivo do logger como determinado por *getEffectiveLevel()*.

getEffectiveLevel()
Indica o nível efeito para esse logger. Se um valor diferente de NOTSET foi definido usando *setLevel()*, ele é retornado. Caso contrário, a hierarquia é percorrida em direção a raiz até um valor diferente de NOTSET ser encontrado e então esse valor é retornado. O valor retornado é um inteiro, tipicamento um entre logging.DEBUG, logging.INFO etc

getChild(*suffix*)
Retorna um logger que é descendente deste logger, como determinado pelo sufixo. Portanto, logging.getLogger('abc').getChild('def.ghi') retornaria o mesmo logger que seria retornado por logging.getLogger('abc.def.ghi'). Esse é um método de conveniência, útil quando o logger pai é nomeado usando, por exemplo __name__ ao invés de uma string literal.

Novo na versão 3.2.

debug(*msg*, **args*, ***kwargs*)
Registra uma mensagem com nível DEBUG neste registrador. O *msg* é a sequência de caracteres do formato da mensagem e os *args* são os argumentos que são mesclados em *msg* usando o operador de formatação da sequência. (Observe que isso significa que você pode usar palavras-chave na string de formato, juntamente com um único argumento do dicionário.)

There are three keyword arguments in *kwargs* which are inspected: *exc_info*, *stack_info*, and *extra*.

If *exc_info* does not evaluate as false, it causes exception information to be added to the logging message. If an exception tuple (in the format returned by *sys.exc_info()*) or an exception instance is provided, it is used; otherwise, *sys.exc_info()* is called to get the exception information.

The second optional keyword argument is *stack_info*, which defaults to False. If true, stack information is added to the logging message, including the actual logging call. Note that this is not the same stack information as that displayed through specifying *exc_info*: The former is stack frames from the bottom of the stack up to the logging call in the current thread, whereas the latter is information about stack frames which have been unwound, following an exception, while searching for exception handlers.

You can specify *stack_info* independently of *exc_info*, e.g. to just show how you got to a certain point in your code, even when no exceptions were raised. The stack frames are printed following a header line which says:

```
Stack (most recent call last):
```

Isso imita o `Traceback (most recent call last):` que é usado ao exibir quadros de exceção.

The third keyword argument is *extra* which can be used to pass a dictionary which is used to populate the __dict__ of the LogRecord created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

```
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset', extra=d)
```

imprimiria algo como

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection␣
↪reset
```

The keys in the dictionary passed in *extra* should not clash with the keys used by the logging system. (See the *Formatter* documentation for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the *Formatter* has been set up with a format string which expects 'clientip' and 'user' in the attribute dictionary of the LogRecord. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case, you always need to pass the *extra* dictionary with these keys.

While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-threaded servers where the same code executes in many contexts, and interesting conditions which arise are dependent on this context (such as remote client IP address and authenticated user name, in the above example). In such circumstances, it is likely that specialized *Formatter*s would be used with particular *Handler*s.

Novo na versão 3.2: The *stack_info* parameter was added.

Alterado na versão 3.5: The *exc_info* parameter can now accept exception instances.

info(*msg*, **args*, ***kwargs*)
Logs a message with level `INFO` on this logger. The arguments are interpreted as for *debug()*.

warning(*msg*, **args*, ***kwargs*)
Logs a message with level `WARNING` on this logger. The arguments are interpreted as for *debug()*.

Nota: There is an obsolete method `warn` which is functionally identical to `warning`. As `warn` is deprecated, please do not use it - use `warning` instead.

---

error(*msg*, **args*, ***kwargs*)
Logs a message with level `ERROR` on this logger. The arguments are interpreted as for *debug()*.

critical(*msg*, **args*, ***kwargs*)
Logs a message with level `CRITICAL` on this logger. The arguments are interpreted as for *debug()*.

log(*level*, *msg*, **args*, ***kwargs*)
Logs a message with integer level *level* on this logger. The other arguments are interpreted as for *debug()*.

exception(*msg*, **args*, ***kwargs*)
Logs a message with level `ERROR` on this logger. The arguments are interpreted as for *debug()*. Exception info is added to the logging message. This method should only be called from an exception handler.

addFilter(*filter*)
Adds the specified filter *filter* to this logger.

removeFilter(*filter*)
Removes the specified filter *filter* from this logger.

filter(*record*)
Apply this logger's filters to the record and return `True` if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be processed (passed to handlers). If one returns a false value, no further processing of the record occurs.

addHandler(*hdlr*)
Adiciona o tratador especificado por *hdlr* deste logger.

removeHandler(*hdlr*)
Remove o tratador especificado por *hdlr* deste logger.

findCaller(*stack_info=False*)
Finds the caller's source filename and line number. Returns the filename, line number, function name and stack information as a 4-element tuple. The stack information is returned as `None` unless *stack_info* is `True`.

handle(*record*)
Handles a record by passing it to all handlers associated with this logger and its ancestors (until a false value of *propagate* is found). This method is used for unpickled records received from a socket, as well as those created locally. Logger-level filtering is applied using *filter()*.

makeRecord(*name*, *level*, *fn*, *lno*, *msg*, *args*, *exc_info*, *func=None*, *extra=None*, *sinfo=None*)
This is a factory method which can be overridden in subclasses to create specialized *LogRecord* instances.

hasHandlers()
Checks to see if this logger has any handlers configured. This is done by looking for handlers in this logger and its parents in the logger hierarchy. Returns `True` if a handler was found, else `False`. The method stops searching up the hierarchy whenever a logger with the 'propagate' attribute set to false is found - that will be the last logger which is checked for the existence of handlers.

Novo na versão 3.2.

Alterado na versão 3.7: Loggers can now be pickled and unpickled.

16.6.2 Logging Levels

The numeric values of logging levels are given in the following table. These are primarily of interest if you want to define your own levels, and need them to have specific values relative to the predefined levels. If you define a level with the same numeric value, it overwrites the predefined value; the predefined name is lost.

Nível	Valor numérico
`CRITICAL`	50
`ERROR`	40
`WARNING`	30
`INFO`	20
`DEBUG`	10
`NOTSET`	0

16.6.3 Manipulação de Objetos

Handlers have the following attributes and methods. Note that *Handler* is never instantiated directly; this class acts as a base for more useful subclasses. However, the `__init__()` method in subclasses needs to call *Handler.__init__()*.

class logging.**Handler**

__init__(*level=NOTSET*)
Initializes the *Handler* instance by setting its level, setting the list of filters to the empty list and creating a lock (using *createLock()*) for serializing access to an I/O mechanism.

createLock()
Initializes a thread lock which can be used to serialize access to underlying I/O functionality which may not be threadsafe.

acquire()
Acquires the thread lock created with *createLock()*.

release()
Releases the thread lock acquired with *acquire()*.

setLevel(*level*)
Sets the threshold for this handler to *level*. Logging messages which are less severe than *level* will be ignored. When a handler is created, the level is set to `NOTSET` (which causes all messages to be processed).

Veja *Logging Levels* para uma lista de níveis.

Alterado na versão 3.2: The *level* parameter now accepts a string representation of the level such as 'INFO' as an alternative to the integer constants such as `INFO`.

setFormatter(*fmt*)
Sets the *Formatter* for this handler to *fmt*.

addFilter(*filter*)
Adds the specified filter *filter* to this handler.

removeFilter(*filter*)
Removes the specified filter *filter* from this handler.

filter(*record*)
Apply this handler's filters to the record and return `True` if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be emitted. If one returns a false value, the handler will not emit the record.

flush()
Ensure all logging output has been flushed. This version does nothing and is intended to be implemented by subclasses.

close()
Tidy up any resources used by the handler. This version does no output but removes the handler from an internal list of handlers which is closed when *shutdown()* is called. Subclasses should ensure that this gets called from overridden *close()* methods.

handle(*record*)
Conditionally emits the specified logging record, depending on filters which may have been added to the handler. Wraps the actual emission of the record with acquisition/release of the I/O thread lock.

handleError(*record*)
This method should be called from handlers when an exception is encountered during an *emit()* call. If the module-level attribute `raiseExceptions` is `False`, exceptions get silently ignored. This is what is

mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The specified record is the one which was being processed when the exception occurred. (The default value of `raiseExceptions` is `True`, as that is more useful during development).

format(*record*)
Do formatting for a record - if a formatter is set, use it. Otherwise, use the default formatter for the module.

emit(*record*)
Do whatever it takes to actually log the specified logging record. This version is intended to be implemented by subclasses and so raises a *NotImplementedError*.

For a list of handlers included as standard, see *logging.handlers*.

16.6.4 Formatter Objects

Formatter objects have the following attributes and methods. They are responsible for converting a *LogRecord* to (usually) a string which can be interpreted by either a human or an external system. The base *Formatter* allows a formatting string to be specified. If none is supplied, the default value of `'%(message)s'` is used, which just includes the message in the logging call. To have additional items of information in the formatted output (such as a timestamp), keep reading.

A Formatter can be initialized with a format string which makes use of knowledge of the *LogRecord* attributes - such as the default value mentioned above making use of the fact that the user's message and arguments are pre-formatted into a *LogRecord*'s *message* attribute. This format string contains standard Python %-style mapping keys. See section *Formatação de String no Formato printf-style* for more information on string formatting.

The useful mapping keys in a *LogRecord* are given in the section on *Atributos LogRecord*.

class logging.**Formatter**(*fmt=None*, *datefmt=None*, *style='%'*)
Returns a new instance of the *Formatter* class. The instance is initialized with a format string for the message as a whole, as well as a format string for the date/time portion of a message. If no *fmt* is specified, `'%(message)s'` is used. If no *datefmt* is specified, a format is used which is described in the *formatTime()* documentation.

The *style* parameter can be one of '%', '{' or '$' and determines how the format string will be merged with its data: using one of %-formatting, *str.format()* or *string.Template*. See formatting-styles for more information on using {- and $-formatting for log messages.

Alterado na versão 3.2: The *style* parameter was added.

format(*record*)
O dicionário de atributo do registro é usado como operando para uma operação de formatação de string. Retorna a sequência resultante. Antes de formatar o dicionário, são executadas algumas etapas preparatórias. O atributo *message* do registro é calculado usando *msg % args*. Se a string de formatação contiver `'(asctime)'`, *formatTime()* será chamado para formatar a hora do evento. Se houver informações de exceção, elas serão formatadas usando *formatException()* e anexadas à mensagem. Observe que as informações da exceção formatada são armazenadas em cache no atributo *exc_text*. Isso é útil porque as informações de exceção podem ser selecionadas e enviadas através da conexão, mas você deve ter cuidado se tiver mais de uma subclasse *Formatter*, que personaliza a formatação das informações de exceção. Nesse caso, você terá que limpar o valor em cache após a formatação de um formatador, para que o próximo formatador para manipular o evento não use o valor em cache, mas o recalcule novamente.

If stack information is available, it's appended after the exception information, using *formatStack()* to transform it if necessary.

formatTime(*record*, *datefmt=None*)
This method should be called from *format()* by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behavior

is as follows: if *datefmt* (a string) is specified, it is used with `time.strftime()` to format the creation time of the record. Otherwise, the format '%Y-%m-%d %H:%M:%S,uuu' is used, where the uuu part is a millisecond value and the other letters are as per the `time.strftime()` documentation. An example time in this format is `2003-01-23 00:29:50,411`. The resulting string is returned.

This function uses a user-configurable function to convert the creation time to a tuple. By default, `time.localtime()` is used; to change this for a particular formatter instance, set the `converter` attribute to a function with the same signature as `time.localtime()` or `time.gmtime()`. To change it for all formatters, for example if you want all logging times to be shown in GMT, set the `converter` attribute in the `Formatter` class.

Alterado na versão 3.3: Previously, the default format was hard-coded as in this example: `2010-09-06 22:38:15,292` where the part before the comma is handled by a strptime format string (`'%Y-%m-%d %H:%M:%S'`), and the part after the comma is a millisecond value. Because strptime does not have a format placeholder for milliseconds, the millisecond value is appended using another format string, `'%s,%03d'` — and both of these format strings have been hardcoded into this method. With the change, these strings are defined as class-level attributes which can be overridden at the instance level when desired. The names of the attributes are `default_time_format` (for the strptime format string) and `default_msec_format` (for appending the millisecond value).

formatException(*exc_info*)
Formats the specified exception information (a standard exception tuple as returned by `sys.exc_info()`) as a string. This default implementation just uses `traceback.print_exception()`. The resulting string is returned.

formatStack(*stack_info*)
Formats the specified stack information (a string as returned by `traceback.print_stack()`, but with the last newline removed) as a string. This default implementation just returns the input value.

16.6.5 Filter Objects

`Filters` can be used by `Handlers` and `Loggers` for more sophisticated filtering than is provided by levels. The base filter class only allows events which are below a certain point in the logger hierarchy. For example, a filter initialized with 'A.B' will allow events logged by loggers 'A.B', 'A.B.C', 'A.B.C.D', 'A.B.D' etc. but not 'A.BB', 'B.A.B' etc. If initialized with the empty string, all events are passed.

class logging.**Filter**(*name=''*)
Returns an instance of the `Filter` class. If *name* is specified, it names a logger which, together with its children, will have its events allowed through the filter. If *name* is the empty string, allows every event.

filter(*record*)
Is the specified record to be logged? Returns zero for no, nonzero for yes. If deemed appropriate, the record may be modified in-place by this method.

Note that filters attached to handlers are consulted before an event is emitted by the handler, whereas filters attached to loggers are consulted whenever an event is logged (using `debug()`, `info()`, etc.), before sending an event to handlers. This means that events which have been generated by descendant loggers will not be filtered by a logger's filter setting, unless the filter has also been applied to those descendant loggers.

You don't actually need to subclass `Filter`: you can pass any instance which has a `filter` method with the same semantics.

Alterado na versão 3.2: You don't need to create specialized `Filter` classes, or use other classes with a `filter` method: you can use a function (or other callable) as a filter. The filtering logic will check to see if the filter object has a `filter` attribute: if it does, it's assumed to be a `Filter` and its `filter()` method is called. Otherwise, it's assumed to be a callable and called with the record as the single parameter. The returned value should conform to that returned by `filter()`.

Although filters are used primarily to filter records based on more sophisticated criteria than levels, they get to see every record which is processed by the handler or logger they're attached to: this can be useful if you want to do things like counting how many records were processed by a particular logger or handler, or adding, changing or removing attributes in the LogRecord being processed. Obviously changing the LogRecord needs to be done with some care, but it does allow the injection of contextual information into logs (see filters-contextual).

16.6.6 LogRecord Objects

LogRecord instances are created automatically by the *Logger* every time something is logged, and can be created manually via *makeLogRecord()* (for example, from a pickled event received over the wire).

class logging.**LogRecord**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*, *sinfo=None*)

Contains all the information pertinent to the event being logged.

The primary information is passed in `msg` and `args`, which are combined using `msg % args` to create the `message` field of the record.

Parâmetros

- **name** – The name of the logger used to log the event represented by this LogRecord. Note that this name will always have this value, even though it may be emitted by a handler attached to a different (ancestor) logger.
- **level** – The numeric level of the logging event (one of DEBUG, INFO etc.) Note that this is converted to *two* attributes of the LogRecord: `levelno` for the numeric value and `levelname` for the corresponding level name.
- **pathname** – The full pathname of the source file where the logging call was made.
- **lineno** – The line number in the source file where the logging call was made.
- **msg** – The event description message, possibly a format string with placeholders for variable data.
- **args** – Variable data to merge into the *msg* argument to obtain the event description.
- **exc_info** – An exception tuple with the current exception information, or `None` if no exception information is available.
- **func** – The name of the function or method from which the logging call was invoked.
- **sinfo** – A text string representing stack information from the base of the stack in the current thread, up to the logging call.

getMessage()

Returns the message for this *LogRecord* instance after merging any user-supplied arguments with the message. If the user-supplied message argument to the logging call is not a string, *str()* is called on it to convert it to a string. This allows use of user-defined classes as messages, whose `__str__` method can return the actual format string to be used.

Alterado na versão 3.2: The creation of a `LogRecord` has been made more configurable by providing a factory which is used to create the record. The factory can be set using *getLogRecordFactory()* and *setLogRecordFactory()* (see this for the factory's signature).

This functionality can be used to inject your own values into a LogRecord at creation time. You can use the following pattern:

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
```

(continua na próxima página)

(continuação da página anterior)

```
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

With this pattern, multiple factories could be chained, and as long as they don't overwrite each other's attributes or unintentionally overwrite the standard attributes listed above, there should be no surprises.

16.6.7 Atributos LogRecord

The LogRecord has a number of attributes, most of which are derived from the parameters to the constructor. (Note that the names do not always correspond exactly between the LogRecord constructor parameters and the LogRecord attributes.) These attributes can be used to merge data from the record into the format string. The following table lists (in alphabetical order) the attribute names, their meanings and the corresponding placeholder in a %-style format string.

If you are using {}-formatting (*str.format()*), you can use `{attrname}` as the placeholder in the format string. If you are using $-formatting (*string.Template*), use the form `${attrname}`. In both cases, of course, replace `attrname` with the actual attribute name you want to use.

In the case of {}-formatting, you can specify formatting flags by placing them after the attribute name, separated from it with a colon. For example: a placeholder of `{msecs:03d}` would format a millisecond value of `4` as `004`. Refer to the *str.format()* documentation for full details on the options available to you.

Attribute name	Formatação	Description (descrição)
args	You shouldn't need to format this yourself.	The tuple of arguments merged into `msg` to produce `message`, or a dict whose values are used for the merge (when there is only one argument, and it is a dictionary).
asctime	`%(asctime)s`	Human-readable time when the *LogRecord* was created. By default this is of the form '2003-07-08 16:49:45,896' (the numbers after the comma are millisecond portion of the time).
created	`%(created)f`	Time when the *LogRecord* was created (as returned by *time.time()*).
exc_info	You shouldn't need to format this yourself.	Exception tuple (à la `sys.exc_info`) or, if no exception has occurred, `None`.
filename	`%(filename)s`	Filename portion of `pathname`.
funcName	`%(funcName)s`	Name of function containing the logging call.
levelname	`%(levelname)s`	Text logging level for the message (`'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`).
levelno	`%(levelno)s`	Numeric logging level for the message (`DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`).
lineno	`%(lineno)d`	Source line number where the logging call was issued (if available).
message	`%(message)s`	The logged message, computed as `msg % args`. This is set when *Formatter.format()* is invoked.
módulo	`%(module)s`	Module (name portion of `filename`).
msecs	`%(msecs)d`	Millisecond portion of the time when the *LogRecord* was created.
msg	You shouldn't need to format this yourself.	The format string passed in the original logging call. Merged with `args` to produce `message`, or an arbitrary object (see arbitrary-object-messages).
nome	`%(name)s`	Name of the logger used to log the call.
pathname	`%(pathname)s`	Full pathname of the source file where the logging call was issued (if available).
process	`%(process)d`	Process ID (if available).
processName	`%(processName)s`	Process name (if available).
relativeCreated	`%(relativeCreated)d`	Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded.
stack_info	You shouldn't need to format this yourself.	Stack frame information (where available) from the bottom of the stack in the current thread, up to and including the stack frame of the logging call which resulted in the creation of this record.
thread	`%(thread)d`	Thread ID (if available).
threadName	`%(threadName)s`	Thread name (if available).

Alterado na versão 3.1: *processName* was added.

16.6.8 LoggerAdapter Objects

LoggerAdapter instances are used to conveniently pass contextual information into logging calls. For a usage example, see the section on adding contextual information to your logging output.

class logging.**LoggerAdapter**(*logger*, *extra*)
Returns an instance of *LoggerAdapter* initialized with an underlying *Logger* instance and a dict-like object.

process(*msg*, *kwargs*)
Modifies the message and/or keyword arguments passed to a logging call in order to insert contextual information. This implementation takes the object passed as *extra* to the constructor and adds it to *kwargs* using key 'extra'. The return value is a (*msg*, *kwargs*) tuple which has the (possibly modified) versions of the arguments passed in.

In addition to the above, *LoggerAdapter* supports the following methods of *Logger*: *debug()*, *info()*, *warning()*, *error()*, *exception()*, *critical()*, *log()*, *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()* and *hasHandlers()*. These methods have the same signatures as their counterparts in *Logger*, so you can use the two types of instances interchangeably.

Alterado na versão 3.2: The *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()* and *hasHandlers()* methods were added to *LoggerAdapter*. These methods delegate to the underlying logger.

16.6.9 Thread Safety

The logging module is intended to be thread-safe without any special work needing to be done by its clients. It achieves this though using threading locks; there is one lock to serialize access to the module's shared data, and each handler also creates a lock to serialize access to its underlying I/O.

If you are implementing asynchronous signal handlers using the *signal* module, you may not be able to use logging from within such handlers. This is because lock implementations in the *threading* module are not always re-entrant, and so cannot be invoked from such signal handlers.

16.6.10 Funções de Nível de Módulo

In addition to the classes described above, there are a number of module-level functions.

logging.**getLogger**(*name=None*)
Return a logger with the specified name or, if name is None, return a logger which is the root logger of the hierarchy. If specified, the name is typically a dot-separated hierarchical name like *'a'*, *'a.b'* or *'a.b.c.d'*. Choice of these names is entirely up to the developer who is using logging.

All calls to this function with a given name return the same logger instance. This means that logger instances never need to be passed between different parts of an application.

logging.**getLoggerClass**()
Return either the standard *Logger* class, or the last class passed to *setLoggerClass()*. This function may be called from within a new class definition, to ensure that installing a customized *Logger* class will not undo customizations already applied by other code. For example:

```
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

logging.**getLogRecordFactory**()
Return a callable which is used to create a *LogRecord*.

Novo na versão 3.2: This function has been provided, along with *setLogRecordFactory()*, to allow developers more control over how the *LogRecord* representing a logging event is constructed.

See *setLogRecordFactory()* for more information about the how the factory is called.

logging.**debug**(*msg*, **args*, ***kwargs*)

Logs a message with level DEBUG on the root logger. The *msg* is the message format string, and the *args* are the arguments which are merged into *msg* using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)

There are three keyword arguments in *kwargs* which are inspected: *exc_info* which, if it does not evaluate as false, causes exception information to be added to the logging message. If an exception tuple (in the format returned by *sys.exc_info()*) or an exception instance is provided, it is used; otherwise, *sys.exc_info()* is called to get the exception information.

The second optional keyword argument is *stack_info*, which defaults to False. If true, stack information is added to the logging message, including the actual logging call. Note that this is not the same stack information as that displayed through specifying *exc_info*: The former is stack frames from the bottom of the stack up to the logging call in the current thread, whereas the latter is information about stack frames which have been unwound, following an exception, while searching for exception handlers.

You can specify *stack_info* independently of *exc_info*, e.g. to just show how you got to a certain point in your code, even when no exceptions were raised. The stack frames are printed following a header line which says:

```
Stack (most recent call last):
```

Isso imita o Traceback (most recent call last): que é usado ao exibir quadros de exceção.

The third optional keyword argument is *extra* which can be used to pass a dictionary which is used to populate the __dict__ of the LogRecord created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

```
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logging.warning('Protocol problem: %s', 'connection reset', extra=d)
```

would print something like:

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection reset
```

The keys in the dictionary passed in *extra* should not clash with the keys used by the logging system. (See the *Formatter* documentation for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the *Formatter* has been set up with a format string which expects 'clientip' and 'user' in the attribute dictionary of the LogRecord. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case, you always need to pass the *extra* dictionary with these keys.

While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-threaded servers where the same code executes in many contexts, and interesting conditions which arise are dependent on this context (such as remote client IP address and authenticated user name, in the above example). In such circumstances, it is likely that specialized *Formatter*s would be used with particular *Handler*s.

Novo na versão 3.2: The *stack_info* parameter was added.

logging.**info**(*msg*, **args*, ***kwargs*)

Logs a message with level INFO on the root logger. The arguments are interpreted as for *debug()*.

logging.**warning**(*msg*, **args*, ***kwargs*)

Logs a message with level WARNING on the root logger. The arguments are interpreted as for *debug()*.

Nota: There is an obsolete function `warn` which is functionally identical to `warning`. As `warn` is deprecated, please do not use it - use `warning` instead.

`logging.`**`error`**(*msg*, **args*, ***kwargs*)
Logs a message with level `ERROR` on the root logger. The arguments are interpreted as for *debug()*.

`logging.`**`critical`**(*msg*, **args*, ***kwargs*)
Logs a message with level `CRITICAL` on the root logger. The arguments are interpreted as for *debug()*.

`logging.`**`exception`**(*msg*, **args*, ***kwargs*)
Logs a message with level `ERROR` on the root logger. The arguments are interpreted as for *debug()*. Exception info is added to the logging message. This function should only be called from an exception handler.

`logging.`**`log`**(*level*, *msg*, **args*, ***kwargs*)
Logs a message with level *level* on the root logger. The other arguments are interpreted as for *debug()*.

Nota: The above module-level convenience functions, which delegate to the root logger, call *basicConfig()* to ensure that at least one handler is available. Because of this, they should *not* be used in threads, in versions of Python earlier than 2.7.1 and 3.2, unless at least one handler has been added to the root logger *before* the threads are started. In earlier versions of Python, due to a thread safety shortcoming in *basicConfig()*, this can (under rare circumstances) lead to handlers being added multiple times to the root logger, which can in turn lead to multiple messages for the same event.

`logging.`**`disable`**(*level=CRITICAL*)
Provides an overriding level *level* for all loggers which takes precedence over the logger's own level. When the need arises to temporarily throttle logging output down across the whole application, this function can be useful. Its effect is to disable all logging calls of severity *level* and below, so that if you call it with a value of INFO, then all INFO and DEBUG events would be discarded, whereas those of severity WARNING and above would be processed according to the logger's effective level. If `logging.disable(logging.NOTSET)` is called, it effectively removes this overriding level, so that logging output again depends on the effective levels of individual loggers.

Note that if you have defined any custom logging level higher than `CRITICAL` (this is not recommended), you won't be able to rely on the default value for the *level* parameter, but will have to explicitly supply a suitable value.

Alterado na versão 3.7: The *level* parameter was defaulted to level `CRITICAL`. See Issue #28524 for more information about this change.

`logging.`**`addLevelName`**(*level*, *levelName*)
Associates level *level* with text *levelName* in an internal dictionary, which is used to map numeric levels to a textual representation, for example when a *Formatter* formats a message. This function can also be used to define your own levels. The only constraints are that all levels used must be registered using this function, levels should be positive integers and they should increase in increasing order of severity.

Nota: If you are thinking of defining your own levels, please see the section on custom-levels.

`logging.`**`getLevelName`**(*level*)
Returns the textual representation of logging level *level*. If the level is one of the predefined levels `CRITICAL`, `ERROR`, `WARNING`, `INFO` or `DEBUG` then you get the corresponding string. If you have associated levels with names using *addLevelName()* then the name you have associated with *level* is returned. If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned. Otherwise, the string 'Level %s' % level is returned.

Nota: Levels are internally integers (as they need to be compared in the logging logic). This function is used to convert between an integer level and the level name displayed in the formatted log output by means of the `%(levelname)s` format specifier (see *Atributos LogRecord*).

Alterado na versão 3.4: In Python versions earlier than 3.4, this function could also be passed a text level, and would return the corresponding numeric value of the level. This undocumented behaviour was considered a mistake, and was removed in Python 3.4, but reinstated in 3.4.2 due to retain backward compatibility.

`logging.`**`makeLogRecord`**`(`*attrdict*`)`

Creates and returns a new `LogRecord` instance whose attributes are defined by *attrdict*. This function is useful for taking a pickled `LogRecord` attribute dictionary, sent over a socket, and reconstituting it as a `LogRecord` instance at the receiving end.

`logging.`**`basicConfig`**`(`***kwargs*`)`

Does basic configuration for the logging system by creating a `StreamHandler` with a default `Formatter` and adding it to the root logger. The functions `debug()`, `info()`, `warning()`, `error()` and `critical()` will call `basicConfig()` automatically if no handlers are defined for the root logger.

This function does nothing if the root logger already has handlers configured for it.

Nota: This function should be called from the main thread before other threads are started. In versions of Python prior to 2.7.1 and 3.2, if this function is called from multiple threads, it is possible (in rare circumstances) that a handler will be added to the root logger more than once, leading to unexpected results such as messages being duplicated in the log.

The following keyword arguments are supported.

Formatação	Description (descrição)
filename	Specifies that a FileHandler be created, using the specified filename, rather than a StreamHandler.
filemode	If *filename* is specified, open the file in this *mode*. Defaults to `'a'`.
format	Use the specified format string for the handler.
datefmt	Use the specified date/time format, as accepted by `time.strftime()`.
style	If *format* is specified, use this style for the format string. One of `'%'`, `'{'` or `'$'` for *printf-style*, `str.format()` or `string.Template` respectively. Defaults to `'%'`.
level	Set the root logger level to the specified *level*.
stream	Use the specified stream to initialize the StreamHandler. Note that this argument is incompatible with *filename* - if both are present, a `ValueError` is raised.
handlers	If specified, this should be an iterable of already created handlers to add to the root logger. Any handlers which don't already have a formatter set will be assigned the default formatter created in this function. Note that this argument is incompatible with *filename* or *stream* - if both are present, a `ValueError` is raised.

Alterado na versão 3.2: The *style* argument was added.

Alterado na versão 3.3: The *handlers* argument was added. Additional checks were added to catch situations where incompatible arguments are specified (e.g. *handlers* together with *stream* or *filename*, or *stream* together with *filename*).

`logging.`**`shutdown`**`()`

Informs the logging system to perform an orderly shutdown by flushing and closing all handlers. This should be called at application exit and no further use of the logging system should be made after this call.

logging.**setLoggerClass**(*klass*)
Tells the logging system to use the class *klass* when instantiating a logger. The class should define `__init__()` such that only a name argument is required, and the `__init__()` should call `Logger.__init__()`. This function is typically called before any loggers are instantiated by applications which need to use custom logger behavior. After this call, as at any other time, do not instantiate loggers directly using the subclass: continue to use the *logging.getLogger()* API to get your loggers.

logging.**setLogRecordFactory**(*factory*)
Set a callable which is used to create a *LogRecord*.

Parâmetros **factory** – The factory callable to be used to instantiate a log record.

Novo na versão 3.2: This function has been provided, along with *getLogRecordFactory()*, to allow developers more control over how the *LogRecord* representing a logging event is constructed.

The factory has the following signature:

```
factory(name, level, fn, lno, msg, args, exc_info, func=None, sinfo=None,
**kwargs)
```

nome The logger name.

level The logging level (numeric).

fn The full pathname of the file where the logging call was made.

lno The line number in the file where the logging call was made.

msg The logging message.

args The arguments for the logging message.

exc_info An exception tuple, or `None`.

func The name of the function or method which invoked the logging call.

sinfo A stack traceback such as is provided by *traceback.print_stack()*, showing the call hierarchy.

kwargs Additional keyword arguments.

16.6.11 Module-Level Attributes

logging.**lastResort**
A "handler of last resort" is available through this attribute. This is a *StreamHandler* writing to `sys.stderr` with a level of `WARNING`, and is used to handle logging events in the absence of any logging configuration. The end result is to just print the message to `sys.stderr`. This replaces the earlier error message saying that "no handlers could be found for logger XYZ". If you need the earlier behaviour for some reason, `lastResort` can be set to `None`.

Novo na versão 3.2.

16.6.12 Integration with the warnings module

The *captureWarnings()* function can be used to integrate *logging* with the *warnings* module.

logging.**captureWarnings**(*capture*)
This function is used to turn the capture of warnings by logging on and off.

If *capture* is True, warnings issued by the *warnings* module will be redirected to the logging system. Specifically, a warning will be formatted using *warnings.formatwarning()* and the resulting string logged to a logger named 'py.warnings' with a severity of WARNING.

If *capture* is False, the redirection of warnings to the logging system will stop, and warnings will be redirected to their original destinations (i.e. those in effect before captureWarnings(True) was called).

Ver também:

Módulo *logging.config* API de configuração para o módulo logging.

Módulo *logging.handlers* Useful handlers included with the logging module.

PEP 282 - A Logging System The proposal which described this feature for inclusion in the Python standard library.

'Original Python logging package '_ This is the original source for the *logging* package. The version of the package available from this site is suitable for use with Python 1.5.2, 2.1.x and 2.2.x, which do not include the *logging* package in the standard library.

16.7 logging.config — Logging configuration

Código Fonte: Lib/logging/config.py

Important

This page contains only reference information. For tutorials, please see

- Basic Tutorial
- Advanced Tutorial
- Logging Cookbook

This section describes the API for configuring the logging module.

16.7.1 Configuration functions

The following functions configure the logging module. They are located in the *logging.config* module. Their use is optional — you can configure the logging module using these functions or by making calls to the main API (defined in *logging* itself) and defining handlers which are declared either in *logging* or *logging.handlers*.

logging.config.**dictConfig**(*config*)

Takes the logging configuration from a dictionary. The contents of this dictionary are described in *Configuration dictionary schema* below.

If an error is encountered during configuration, this function will raise a *ValueError*, *TypeError*, *AttributeError* or *ImportError* with a suitably descriptive message. The following is a (possibly incomplete) list of conditions which will raise an error:

- A `level` which is not a string or which is a string not corresponding to an actual logging level.
- A `propagate` value which is not a boolean.
- An id which does not have a corresponding destination.
- A non-existent handler id found during an incremental call.
- An invalid logger name.
- Inability to resolve to an internal or external object.

Parsing is performed by the `DictConfigurator` class, whose constructor is passed the dictionary used for configuration, and has a `configure()` method. The *logging.config* module has a callable attribute `dictConfigClass` which is initially set to `DictConfigurator`. You can replace the value of `dictConfigClass` with a suitable implementation of your own.

dictConfig() calls `dictConfigClass` passing the specified dictionary, and then calls the `configure()` method on the returned object to put the configuration into effect:

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

For example, a subclass of `DictConfigurator` could call `DictConfigurator.__init__()` in its own `__init__()`, then set up custom prefixes which would be usable in the subsequent `configure()` call. `dictConfigClass` would be bound to this new subclass, and then *dictConfig()* could be called exactly as in the default, uncustomized state.

Novo na versão 3.2.

`logging.config.`**`fileConfig`**(*fname*, *defaults=None*, *disable_existing_loggers=True*)

Reads the logging configuration from a *configparser*-format file. The format of the file should be as described in *Formato do arquivo de configuração*. This function can be called several times from an application, allowing an end user to select from various pre-canned configurations (if the developer provides a mechanism to present the choices and load the chosen configuration).

Parâmetros

- **`fname`** – A filename, or a file-like object, or an instance derived from *RawConfigParser*. If a `RawConfigParser`-derived instance is passed, it is used as is. Otherwise, a `Configparser` is instantiated, and the configuration read by it from the object passed in `fname`. If that has a *readline()* method, it is assumed to be a file-like object and read using *read_file()*; otherwise, it is assumed to be a filename and passed to *read()*.
- **`defaults`** – Defaults to be passed to the ConfigParser can be specified in this argument.
- **`disable_existing_loggers`** – If specified as `False`, loggers which exist when this call is made are left enabled. The default is `True` because this enables old behaviour in a backward-compatible way. This behaviour is to disable any existing non-root loggers unless they or their ancestors are explicitly named in the logging configuration.

Alterado na versão 3.4: An instance of a subclass of *RawConfigParser* is now accepted as a value for `fname`. This facilitates:

- Use of a configuration file where logging configuration is just part of the overall application configuration.
- Use of a configuration read from a file, and then modified by the using application (e.g. based on command-line parameters or other aspects of the runtime environment) before being passed to `fileConfig`.

`logging.config.`**`listen`**(*port=DEFAULT_LOGGING_CONFIG_PORT*, *verify=None*)

Starts up a socket server on the specified port, and listens for new configurations. If no port is specified, the module's default `DEFAULT_LOGGING_CONFIG_PORT` is used. Logging configurations will be sent as a file suitable for processing by *dictConfig()* or *fileConfig()*. Returns a *Thread* instance on which you

can call *start()* to start the server, and which you can *join()* when appropriate. To stop the server, call *stopListening()*.

The `verify` argument, if specified, should be a callable which should verify whether bytes received across the socket are valid and should be processed. This could be done by encrypting and/or signing what is sent across the socket, such that the `verify` callable can perform signature verification and/or decryption. The `verify` callable is called with a single argument - the bytes received across the socket - and should return the bytes to be processed, or `None` to indicate that the bytes should be discarded. The returned bytes could be the same as the passed in bytes (e.g. when only verification is done), or they could be completely different (perhaps if decryption were performed).

To send a configuration to the socket, read in the configuration file and send it to the socket as a sequence of bytes preceded by a four-byte length string packed in binary using `struct.pack('>L', n)`.

Nota: Because portions of the configuration are passed through *eval()*, use of this function may open its users to a security risk. While the function only binds to a socket on `localhost`, and so does not accept connections from remote machines, there are scenarios where untrusted code could be run under the account of the process which calls *listen()*. Specifically, if the process calling *listen()* runs on a multi-user machine where users cannot trust each other, then a malicious user could arrange to run essentially arbitrary code in a victim user's process, simply by connecting to the victim's *listen()* socket and sending a configuration which runs whatever code the attacker wants to have executed in the victim's process. This is especially easy to do if the default port is used, but not hard even if a different port is used). To avoid the risk of this happening, use the `verify` argument to *listen()* to prevent unrecognised configurations from being applied.

Alterado na versão 3.4: The `verify` argument was added.

Nota: If you want to send configurations to the listener which don't disable existing loggers, you will need to use a JSON format for the configuration, which will use *dictConfig()* for configuration. This method allows you to specify `disable_existing_loggers` as `False` in the configuration you send.

`logging.config.`**`stopListening`**`()`

Stops the listening server which was created with a call to *listen()*. This is typically called before calling `join()` on the return value from *listen()*.

16.7.2 Configuration dictionary schema

Describing a logging configuration requires listing the various objects to create and the connections between them; for example, you may create a handler named 'console' and then say that the logger named 'startup' will send its messages to the 'console' handler. These objects aren't limited to those provided by the *logging* module because you might write your own formatter or handler class. The parameters to these classes may also need to include external objects such as `sys.stderr`. The syntax for describing these objects and connections is defined in *Object connections* below.

Dictionary Schema Details

The dictionary passed to *dictConfig()* must contain the following keys:

- *version* - to be set to an integer value representing the schema version. The only valid value at present is 1, but having this key allows the schema to evolve while still preserving backwards compatibility.

All other keys are optional, but if present they will be interpreted as described below. In all cases below where a 'configuring dict' is mentioned, it will be checked for the special `'()'` key to see if a custom instantiation is required. If so, the mechanism described in *User-defined objects* below is used to create an instance; otherwise, the context is used to determine what to instantiate.

- *formatters* - the corresponding value will be a dict in which each key is a formatter id and each value is a dict describing how to configure the corresponding *Formatter* instance.

  The configuring dict is searched for keys `format` and `datefmt` (with defaults of `None`) and these are used to construct a *Formatter* instance.

- *filters* - the corresponding value will be a dict in which each key is a filter id and each value is a dict describing how to configure the corresponding Filter instance.

  The configuring dict is searched for the key `name` (defaulting to the empty string) and this is used to construct a *logging.Filter* instance.

- *handlers* - the corresponding value will be a dict in which each key is a handler id and each value is a dict describing how to configure the corresponding Handler instance.

  The configuring dict is searched for the following keys:

  - `class` (mandatory). This is the fully qualified name of the handler class.
  - `level` (optional). The level of the handler.
  - `formatter` (optional). The id of the formatter for this handler.
  - `filters` (optional). A list of ids of the filters for this handler.

  All *other* keys are passed through as keyword arguments to the handler's constructor. For example, given the snippet:

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level   : INFO
    filters: [allow_foo]
    stream  : ext://sys.stdout
  file:
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

  the handler with id `console` is instantiated as a *logging.StreamHandler*, using `sys.stdout` as the underlying stream. The handler with id `file` is instantiated as a *logging.handlers.RotatingFileHandler* with the keyword arguments `filename='logconfig.log', maxBytes=1024, backupCount=3`.

- *loggers* - the corresponding value will be a dict in which each key is a logger name and each value is a dict describing how to configure the corresponding Logger instance.

  The configuring dict is searched for the following keys:

  - `level` (optional). The level of the logger.
  - `propagate` (optional). The propagation setting of the logger.
  - `filters` (optional). A list of ids of the filters for this logger.
  - `handlers` (optional). A list of ids of the handlers for this logger.

  The specified loggers will be configured according to the level, propagation, filters and handlers specified.

- *root* - this will be the configuration for the root logger. Processing of the configuration will be as for any logger, except that the `propagate` setting will not be applicable.

- *incremental* - whether the configuration is to be interpreted as incremental to the existing configuration. This value defaults to `False`, which means that the specified configuration replaces the existing configuration with the same semantics as used by the existing *`fileConfig()`* API.

  If the specified value is `True`, the configuration is processed as described in the section on *Incremental Configuration*.

- *disable_existing_loggers* - whether any existing non-root loggers are to be disabled. This setting mirrors the parameter of the same name in *`fileConfig()`*. If absent, this parameter defaults to `True`. This value is ignored if *incremental* is `True`.

Incremental Configuration

It is difficult to provide complete flexibility for incremental configuration. For example, because objects such as filters and formatters are anonymous, once a configuration is set up, it is not possible to refer to such anonymous objects when augmenting a configuration.

Furthermore, there is not a compelling case for arbitrarily altering the object graph of loggers, handlers, filters, formatters at run-time, once a configuration is set up; the verbosity of loggers and handlers can be controlled just by setting levels (and, in the case of loggers, propagation flags). Changing the object graph arbitrarily in a safe way is problematic in a multi-threaded environment; while not impossible, the benefits are not worth the complexity it adds to the implementation.

Thus, when the `incremental` key of a configuration dict is present and is `True`, the system will completely ignore any `formatters` and `filters` entries, and process only the `level` settings in the `handlers` entries, and the `level` and `propagate` settings in the `loggers` and `root` entries.

Using a value in the configuration dict lets configurations to be sent over the wire as pickled dicts to a socket listener. Thus, the logging verbosity of a long-running application can be altered over time with no need to stop and restart the application.

Object connections

The schema describes a set of logging objects - loggers, handlers, formatters, filters - which are connected to each other in an object graph. Thus, the schema needs to represent connections between the objects. For example, say that, once configured, a particular logger has attached to it a particular handler. For the purposes of this discussion, we can say that the logger represents the source, and the handler the destination, of a connection between the two. Of course in the configured objects this is represented by the logger holding a reference to the handler. In the configuration dict, this is done by giving each destination object an id which identifies it unambiguously, and then using the id in the source object's configuration to indicate that a connection exists between the source and the destination object with that id.

So, for example, consider the following YAML snippet:

```
formatters:
  brief:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here
handlers:
  h1: #This is an id
   # configuration of handler with id 'h1' goes here
   formatter: brief
  h2: #This is another id
   # configuration of handler with id 'h2' goes here
   formatter: precise
loggers:
  foo.bar.baz:
```

(continua na próxima página)

(continuação da página anterior)

```
    # other configuration for logger 'foo.bar.baz'
    handlers: [h1, h2]
```

(Note: YAML used here because it's a little more readable than the equivalent Python source form for the dictionary.)

The ids for loggers are the logger names which would be used programmatically to obtain a reference to those loggers, e.g. `foo.bar.baz`. The ids for Formatters and Filters can be any string value (such as `brief`, `precise` above) and they are transient, in that they are only meaningful for processing the configuration dictionary and used to determine connections between objects, and are not persisted anywhere when the configuration call is complete.

The above snippet indicates that logger named `foo.bar.baz` should have two handlers attached to it, which are described by the handler ids `h1` and `h2`. The formatter for `h1` is that described by id `brief`, and the formatter for `h2` is that described by id `precise`.

User-defined objects

The schema supports user-defined objects for handlers, filters and formatters. (Loggers do not need to have different types for different instances, so there is no support in this configuration schema for user-defined logger classes.)

Objects to be configured are described by dictionaries which detail their configuration. In some places, the logging system will be able to infer from the context how an object is to be instantiated, but when a user-defined object is to be instantiated, the system will not know how to do this. In order to provide complete flexibility for user-defined object instantiation, the user needs to provide a 'factory' - a callable which is called with a configuration dictionary and which returns the instantiated object. This is signalled by an absolute import path to the factory being made available under the special key `'()'`. Here's a concrete example:

```
formatters:
  brief:
    format: '%(message)s'
  default:
    format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
  custom:
      (): my.package.customFormatterFactory
      bar: baz
      spam: 99.9
      answer: 42
```

The above YAML snippet defines three formatters. The first, with id `brief`, is a standard *`logging.Formatter`* instance with the specified format string. The second, with id `default`, has a longer format and also defines the time format explicitly, and will result in a *`logging.Formatter`* initialized with those two format strings. Shown in Python source form, the `brief` and `default` formatters have configuration sub-dictionaries:

```
{
  'format' : '%(message)s'
}
```

e:

```
{
  'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
  'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

respectively, and as these dictionaries do not contain the special key `'()'`, the instantiation is inferred from the context: as a result, standard *logging.Formatter* instances are created. The configuration sub-dictionary for the third formatter, with id `custom`, is:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42
}
```

and this contains the special key `'()'`, which means that user-defined instantiation is wanted. In this case, the specified factory callable will be used. If it is an actual callable it will be used directly - otherwise, if you specify a string (as in the example) the actual callable will be located using normal import mechanisms. The callable will be called with the **remaining** items in the configuration sub-dictionary as keyword arguments. In the above example, the formatter with id `custom` will be assumed to be returned by the call:

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

The key `'()'` has been used as the special key because it is not a valid keyword parameter name, and so will not clash with the names of the keyword arguments used in the call. The `'()'` also serves as a mnemonic that the corresponding value is a callable.

Access to external objects

There are times where a configuration needs to refer to objects external to the configuration, for example `sys.stderr`. If the configuration dict is constructed using Python code, this is straightforward, but a problem arises when the configuration is provided via a text file (e.g. JSON, YAML). In a text file, there is no standard way to distinguish `sys.stderr` from the literal string `'sys.stderr'`. To facilitate this distinction, the configuration system looks for certain special prefixes in string values and treat them specially. For example, if the literal string `'ext://sys.stderr'` is provided as a value in the configuration, then the `ext://` will be stripped off and the remainder of the value processed using normal import mechanisms.

The handling of such prefixes is done in a way analogous to protocol handling: there is a generic mechanism to look for prefixes which match the regular expression `^(?P<prefix>[a-z]+)://(?P<suffix>.*)$` whereby, if the `prefix` is recognised, the `suffix` is processed in a prefix-dependent manner and the result of the processing replaces the string value. If the prefix is not recognised, then the string value will be left as-is.

Access to internal objects

As well as external objects, there is sometimes also a need to refer to objects in the configuration. This will be done implicitly by the configuration system for things that it knows about. For example, the string value `'DEBUG'` for a `level` in a logger or handler will automatically be converted to the value `logging.DEBUG`, and the `handlers`, `filters` and `formatter` entries will take an object id and resolve to the appropriate destination object.

However, a more generic mechanism is needed for user-defined objects which are not known to the *logging* module. For example, consider *logging.handlers.MemoryHandler*, which takes a `target` argument which is another handler to delegate to. Since the system already knows about this class, then in the configuration, the given `target` just needs to be the object id of the relevant target handler, and the system will resolve to the handler from the id. If, however, a user defines a `my.package.MyHandler` which has an `alternate` handler, the configuration system would not know that the `alternate` referred to a handler. To cater for this, a generic resolution system allows the user to specify:

```
handlers:
  file:
```

(continua na próxima página)

(continuação da página anterior)

```
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

The literal string `'cfg://handlers.file'` will be resolved in an analogous way to strings with the `ext://` prefix, but looking in the configuration itself rather than the import namespace. The mechanism allows access by dot or by index, in a similar way to that provided by `str.format`. Thus, given the following snippet:

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
      - support_team@domain.tld
      - dev_team@domain.tld
    subject: Houston, we have a problem.
```

in the configuration, the string `'cfg://handlers'` would resolve to the dict with key `handlers`, the string `'cfg://handlers.email` would resolve to the dict with key `email` in the `handlers` dict, and so on. The string `'cfg://handlers.email.toaddrs[1]` would resolve to `'dev_team.domain.tld'` and the string `'cfg://handlers.email.toaddrs[0]'` would resolve to the value `'support_team@domain.tld'`. The `subject` value could be accessed using either `'cfg://handlers.email.subject'` or, equivalently, `'cfg://handlers.email[subject]'`. The latter form only needs to be used if the key contains spaces or non-alphanumeric characters. If an index value consists only of decimal digits, access will be attempted using the corresponding integer value, falling back to the string value if needed.

Given a string `cfg://handlers.myhandler.mykey.123`, this will resolve to `config_dict['handlers']['myhandler']['mykey']['123']`. If the string is specified as `cfg://handlers.myhandler.mykey[123]`, the system will attempt to retrieve the value from `config_dict['handlers']['myhandler']['mykey'][123]`, and fall back to `config_dict['handlers']['myhandler']['mykey']['123']` if that fails.

Import resolution and custom importers

Import resolution, by default, uses the builtin *`__import__()`* function to do its importing. You may want to replace this with your own importing mechanism: if so, you can replace the `importer` attribute of the `DictConfigurator` or its superclass, the `BaseConfigurator` class. However, you need to be careful because of the way functions are accessed from classes via descriptors. If you are using a Python callable to do your imports, and you want to define it at class level rather than instance level, you need to wrap it with *`staticmethod()`*. For example:

```
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

You don't need to wrap with *`staticmethod()`* if you're setting the import callable on a configurator *instance*.

16.7.3 Formato do arquivo de configuração

The configuration file format understood by *fileConfig()* is based on *configparser* functionality. The file must contain sections called [loggers], [handlers] and [formatters] which identify by name the entities of each type which are defined in the file. For each such entity, there is a separate section which identifies how that entity is configured. Thus, for a logger named log01 in the [loggers] section, the relevant configuration details are held in a section [logger_log01]. Similarly, a handler called hand01 in the [handlers] section will have its configuration held in a section called [handler_hand01], while a formatter called form01 in the [formatters] section will have its configuration specified in a section called [formatter_form01]. The root logger configuration must be specified in a section called [logger_root].

Nota: The *fileConfig()* API is older than the *dictConfig()* API and does not provide functionality to cover certain aspects of logging. For example, you cannot configure *Filter* objects, which provide for filtering of messages beyond simple integer levels, using *fileConfig()*. If you need to have instances of *Filter* in your logging configuration, you will need to use *dictConfig()*. Note that future enhancements to configuration functionality will be added to *dictConfig()*, so it's worth considering transitioning to this newer API when it's convenient to do so.

Examples of these sections in the file are given below.

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

The root logger must specify a level and a list of handlers. An example of a root logger section is given below.

```
[logger_root]
level=NOTSET
handlers=hand01
```

The level entry can be one of DEBUG, INFO, WARNING, ERROR, CRITICAL or NOTSET. For the root logger only, NOTSET means that all messages will be logged. Level values are *eval()*uated in the context of the logging package's namespace.

The handlers entry is a comma-separated list of handler names, which must appear in the [handlers] section. These names must appear in the [handlers] section and have corresponding sections in the configuration file.

For loggers other than the root logger, some additional information is required. This is illustrated by the following example.

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

The level and handlers entries are interpreted as for the root logger, except that if a non-root logger's level is specified as NOTSET, the system consults loggers higher up the hierarchy to determine the effective level of the logger. The propagate entry is set to 1 to indicate that messages must propagate to handlers higher up the logger hierarchy from this logger, or 0 to indicate that messages are **not** propagated to handlers up the hierarchy. The qualname entry is the hierarchical channel name of the logger, that is to say the name used by the application to get the logger.

Sections which specify handler configuration are exemplified by the following.

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

The `class` entry indicates the handler's class (as determined by *eval()* in the `logging` package's namespace). The `level` is interpreted as for loggers, and `NOTSET` is taken to mean 'log everything'.

The `formatter` entry indicates the key name of the formatter for this handler. If blank, a default formatter (`logging._defaultFormatter`) is used. If a name is specified, it must appear in the `[formatters]` section and have a corresponding section in the configuration file.

The `args` entry, when *eval()*uated in the context of the `logging` package's namespace, is the list of arguments to the constructor for the handler class. Refer to the constructors for the relevant handlers, or to the examples below, to see how typical entries are constructed. If not provided, it defaults to `()`.

The optional `kwargs` entry, when *eval()*uated in the context of the `logging` package's namespace, is the keyword argument dict to the constructor for the handler class. If not provided, it defaults to `{}`.

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=(('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')
kwargs={'timeout': 10.0}

[handler_hand08]
```

(continua na próxima página)

(continuação da página anterior)

```
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
kwargs={'secure': True}
```

Sections which specify formatter configuration are typified by the following.

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s
datefmt=
class=logging.Formatter
```

The `format` entry is the overall format string, and the `datefmt` entry is the `strftime()`-compatible date/time format string. If empty, the package substitutes something which is almost equivalent to specifying the date format string `'%Y-%m-%d %H:%M:%S'`. This format also specifies milliseconds, which are appended to the result of using the above format string, with a comma separator. An example time in this format is `2003-01-23 00:29:50,411`.

The `class` entry is optional. It indicates the name of the formatter's class (as a dotted module and class name.) This option is useful for instantiating a *`Formatter`* subclass. Subclasses of *`Formatter`* can present exception tracebacks in an expanded or condensed format.

Nota: Due to the use of *`eval()`* as described above, there are potential security risks which result from using the *`listen()`* to send and receive configurations via sockets. The risks are limited to where multiple users with no mutual trust run code on the same machine; see the *`listen()`* documentation for more information.

Ver também:

Módulo *`logging`* Referência da API para o módulo de logging.

Módulo *`logging.handlers`* Useful handlers included with the logging module.

16.8 `logging.handlers` — Tratadores de registro

Código Fonte: Lib/logging/handlers.py

Important

This page contains only reference information. For tutorials, please see

- Basic Tutorial
- Advanced Tutorial
- Logging Cookbook

The following useful handlers are provided in the package. Note that three of the handlers (*StreamHandler*, *FileHandler* and *NullHandler*) are actually defined in the *logging* module itself, but have been documented here along with the other handlers.

16.8.1 StreamHandler

The *StreamHandler* class, located in the core *logging* package, sends logging output to streams such as *sys.stdout*, *sys.stderr* or any file-like object (or, more precisely, any object which supports `write()` and `flush()` methods).

class logging.**StreamHandler**(*stream=None*)

Returns a new instance of the *StreamHandler* class. If *stream* is specified, the instance will use it for logging output; otherwise, *sys.stderr* will be used.

emit(*record*)

If a formatter is specified, it is used to format the record. The record is then written to the stream with a terminator. If exception information is present, it is formatted using *traceback.print_exception()* and appended to the stream.

flush()

Flushes the stream by calling its *flush()* method. Note that the `close()` method is inherited from *Handler* and so does no output, so an explicit *flush()* call may be needed at times.

setStream(*stream*)

Sets the instance's stream to the specified value, if it is different. The old stream is flushed before the new stream is set.

Parâmetros **stream** – The stream that the handler should use.

Retorna the old stream, if the stream was changed, or *None* if it wasn't.

Novo na versão 3.7.

Alterado na versão 3.2: The `StreamHandler` class now has a `terminator` attribute, default value `'\n'`, which is used as the terminator when writing a formatted record to a stream. If you don't want this newline termination, you can set the handler instance's `terminator` attribute to the empty string. In earlier versions, the terminator was hardcoded as `'\n'`.

16.8.2 FileHandler

The *FileHandler* class, located in the core *logging* package, sends logging output to a disk file. It inherits the output functionality from *StreamHandler*.

class logging.**FileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*)

Returns a new instance of the *FileHandler* class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to *emit()*. By default, the file grows indefinitely.

Alterado na versão 3.6: As well as string values, *Path* objects are also accepted for the *filename* argument.

close()

Closes the file.

emit(*record*)

Outputs the record to the file.

16.8.3 NullHandler

Novo na versão 3.1.

The *NullHandler* class, located in the core *logging* package, does not do any formatting or output. It is essentially a 'no-op' handler for use by library developers.

class logging.**NullHandler**
Returns a new instance of the *NullHandler* class.

emit(*record*)
This method does nothing.

handle(*record*)
This method does nothing.

createLock()
This method returns `None` for the lock, since there is no underlying I/O to which access needs to be serialized.

See library-config for more information on how to use *NullHandler*.

16.8.4 WatchedFileHandler

The *WatchedFileHandler* class, located in the *logging.handlers* module, is a `FileHandler` which watches the file it is logging to. If the file changes, it is closed and reopened using the file name.

A file change can happen because of usage of programs such as *newsyslog* and *logrotate* which perform log file rotation. This handler, intended for use under Unix/Linux, watches the file to see if it has changed since the last emit. (A file is deemed to have changed if its device or inode have changed.) If the file has changed, the old file stream is closed, and the file opened to get a new stream.

This handler is not appropriate for use under Windows, because under Windows open log files cannot be moved or renamed - logging opens the files with exclusive locks - and so there is no need for such a handler. Furthermore, *ST_INO* is not supported under Windows; *stat()* always returns zero for this value.

class logging.handlers.**WatchedFileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*)
Returns a new instance of the *WatchedFileHandler* class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to *emit()*. By default, the file grows indefinitely.

Alterado na versão 3.6: As well as string values, *Path* objects are also accepted for the *filename* argument.

reopenIfNeeded()
Checks to see if the file has changed. If it has, the existing stream is flushed and closed and the file opened again, typically as a precursor to outputting the record to the file.

Novo na versão 3.6.

emit(*record*)
Outputs the record to the file, but first calls *reopenIfNeeded()* to reopen the file if it has changed.

16.8.5 BaseRotatingHandler

The *BaseRotatingHandler* class, located in the *logging.handlers* module, is the base class for the rotating file handlers, *RotatingFileHandler* and *TimedRotatingFileHandler*. You should not need to instantiate this class, but it has attributes and methods you may need to override.

class logging.handlers.**BaseRotatingHandler**(*filename*, *mode*, *encoding=None*, *delay=False*)

The parameters are as for FileHandler. The attributes are:

namer

If this attribute is set to a callable, the *rotation_filename()* method delegates to this callable. The parameters passed to the callable are those passed to *rotation_filename()*.

Nota: The namer function is called quite a few times during rollover, so it should be as simple and as fast as possible. It should also return the same output every time for a given input, otherwise the rollover behaviour may not work as expected.

Novo na versão 3.3.

rotator

If this attribute is set to a callable, the *rotate()* method delegates to this callable. The parameters passed to the callable are those passed to *rotate()*.

Novo na versão 3.3.

rotation_filename(*default_name*)

Modify the filename of a log file when rotating.

This is provided so that a custom filename can be provided.

The default implementation calls the 'namer' attribute of the handler, if it's callable, passing the default name to it. If the attribute isn't callable (the default is None), the name is returned unchanged.

Parâmetros **default_name** – The default name for the log file.

Novo na versão 3.3.

rotate(*source*, *dest*)

When rotating, rotate the current log.

The default implementation calls the 'rotator' attribute of the handler, if it's callable, passing the source and dest arguments to it. If the attribute isn't callable (the default is None), the source is simply renamed to the destination.

Parâmetros

- **source** – The source filename. This is normally the base filename, e.g. 'test.log'.
- **dest** – The destination filename. This is normally what the source is rotated to, e.g. 'test.log.1'.

Novo na versão 3.3.

The reason the attributes exist is to save you having to subclass - you can use the same callables for instances of *RotatingFileHandler* and *TimedRotatingFileHandler*. If either the namer or rotator callable raises an exception, this will be handled in the same way as any other exception during an emit() call, i.e. via the handleError() method of the handler.

If you need to make more significant changes to rotation processing, you can override the methods.

For an example, see cookbook-rotator-namer.

16.8.6 RotatingFileHandler

The `RotatingFileHandler` class, located in the `logging.handlers` module, supports rotation of disk log files.

class logging.handlers.**RotatingFileHandler**(*filename*, *mode='a'*, *maxBytes=0*, *backupCount=0*, *encoding=None*, *delay=False*)

Returns a new instance of the `RotatingFileHandler` class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to `emit()`. By default, the file grows indefinitely.

You can use the *maxBytes* and *backupCount* values to allow the file to *rollover* at a predetermined size. When the size is about to be exceeded, the file is closed and a new file is silently opened for output. Rollover occurs whenever the current log file is nearly *maxBytes* in length; but if either of *maxBytes* or *backupCount* is zero, rollover never occurs, so you generally want to set *backupCount* to at least 1, and have a non-zero *maxBytes*. When *backupCount* is non-zero, the system will save old log files by appending the extensions '.1', '.2' etc., to the filename. For example, with a *backupCount* of 5 and a base file name of `app.log`, you would get `app.log`, `app.log.1`, `app.log.2`, up to `app.log.5`. The file being written to is always `app.log`. When this file is filled, it is closed and renamed to `app.log.1`, and if files `app.log.1`, `app.log.2`, etc. exist, then they are renamed to `app.log.2`, `app.log.3` etc. respectively.

Alterado na versão 3.6: As well as string values, `Path` objects are also accepted for the *filename* argument.

doRollover()

Does a rollover, as described above.

emit(*record*)

Outputs the record to the file, catering for rollover as described previously.

16.8.7 TimedRotatingFileHandler

The `TimedRotatingFileHandler` class, located in the `logging.handlers` module, supports rotation of disk log files at certain timed intervals.

class logging.handlers.**TimedRotatingFileHandler**(*filename*, *when='h'*, *interval=1*, *backupCount=0*, *encoding=None*, *delay=False*, *utc=False*, *atTime=None*)

Returns a new instance of the `TimedRotatingFileHandler` class. The specified file is opened and used as the stream for logging. On rotating it also sets the filename suffix. Rotating happens based on the product of *when* and *interval*.

You can use the *when* to specify the type of *interval*. The list of possible values is below. Note that they are not case sensitive.

Valor	Type of interval	If/how *atTime* is used
`'S'`	Seconds	Ignored
`'M'`	Minutes	Ignored
`'H'`	Horas	Ignored
`'D'`	Dias	Ignored
`'W0'-'W6'`	Weekday (0=Monday)	Used to compute initial rollover time
`'midnight'`	Roll over at midnight, if *atTime* not specified, else at time *atTime*	Used to compute initial rollover time

When using weekday-based rotation, specify 'W0' for Monday, 'W1' for Tuesday, and so on up to 'W6' for Sunday. In this case, the value passed for *interval* isn't used.

The system will save old log files by appending extensions to the filename. The extensions are date-and-time based, using the strftime format `%Y-%m-%d_%H-%M-%S` or a leading portion thereof, depending on the rollover interval.

When computing the next rollover time for the first time (when the handler is created), the last modification time of an existing log file, or else the current time, is used to compute when the next rotation will occur.

If the *utc* argument is true, times in UTC will be used; otherwise local time is used.

If *backupCount* is nonzero, at most *backupCount* files will be kept, and if more would be created when rollover occurs, the oldest one is deleted. The deletion logic uses the interval to determine which files to delete, so changing the interval may leave old files lying around.

If *delay* is true, then file opening is deferred until the first call to *emit()*.

If *atTime* is not `None`, it must be a `datetime.time` instance which specifies the time of day when rollover occurs, for the cases where rollover is set to happen "at midnight" or "on a particular weekday". Note that in these cases, the *atTime* value is effectively used to compute the *initial* rollover, and subsequent rollovers would be calculated via the normal interval calculation.

Nota: Calculation of the initial rollover time is done when the handler is initialised. Calculation of subsequent rollover times is done only when rollover occurs, and rollover occurs only when emitting output. If this is not kept in mind, it might lead to some confusion. For example, if an interval of "every minute" is set, that does not mean you will always see log files with times (in the filename) separated by a minute; if, during application execution, logging output is generated more frequently than once a minute, *then* you can expect to see log files with times separated by a minute. If, on the other hand, logging messages are only output once every five minutes (say), then there will be gaps in the file times corresponding to the minutes where no output (and hence no rollover) occurred.

Alterado na versão 3.4: *atTime* parameter was added.

Alterado na versão 3.6: As well as string values, *Path* objects are also accepted for the *filename* argument.

doRollover()
Does a rollover, as described above.

emit(*record*)
Outputs the record to the file, catering for rollover as described above.

16.8.8 SocketHandler

The *SocketHandler* class, located in the *logging.handlers* module, sends logging output to a network socket. The base class uses a TCP socket.

class logging.handlers.**SocketHandler**(*host*, *port*)
Returns a new instance of the *SocketHandler* class intended to communicate with a remote machine whose address is given by *host* and *port*.

Alterado na versão 3.4: If `port` is specified as `None`, a Unix domain socket is created using the value in `host` - otherwise, a TCP socket is created.

close()
Closes the socket.

emit()
Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with

the socket, silently drops the packet. If the connection was previously lost, re-establishes the connection. To unpickle the record at the receiving end into a *LogRecord*, use the *makeLogRecord()* function.

handleError()
Handles an error which has occurred during *emit()*. The most likely cause is a lost connection. Closes the socket so that we can retry on the next event.

makeSocket()
This is a factory method which allows subclasses to define the precise type of socket they want. The default implementation creates a TCP socket (*socket.SOCK_STREAM*).

makePickle(*record*)
Pickles the record's attribute dictionary in binary format with a length prefix, and returns it ready for transmission across the socket. The details of this operation are equivalent to:

```
data = pickle.dumps(record_attr_dict, 1)
datalen = struct.pack('>L', len(data))
return datalen + data
```

Note that pickles aren't completely secure. If you are concerned about security, you may want to override this method to implement a more secure mechanism. For example, you can sign pickles using HMAC and then verify them on the receiving end, or alternatively you can disable unpickling of global objects on the receiving end.

send(*packet*)
Send a pickled byte-string *packet* to the socket. The format of the sent byte-string is as described in the documentation for *makePickle()*.

This function allows for partial sends, which can happen when the network is busy.

createSocket()
Tries to create a socket; on failure, uses an exponential back-off algorithm. On initial failure, the handler will drop the message it was trying to send. When subsequent messages are handled by the same instance, it will not try connecting until some time has passed. The default parameters are such that the initial delay is one second, and if after that delay the connection still can't be made, the handler will double the delay each time up to a maximum of 30 seconds.

This behaviour is controlled by the following handler attributes:

- `retryStart` (initial delay, defaulting to 1.0 seconds).
- `retryFactor` (multiplier, defaulting to 2.0).
- `retryMax` (maximum delay, defaulting to 30.0 seconds).

This means that if the remote listener starts up *after* the handler has been used, you could lose messages (since the handler won't even attempt a connection until the delay has elapsed, but just silently drop messages during the delay period).

16.8.9 DatagramHandler

The *DatagramHandler* class, located in the *logging.handlers* module, inherits from *SocketHandler* to support sending logging messages over UDP sockets.

class logging.handlers.**DatagramHandler**(*host*, *port*)
Returns a new instance of the *DatagramHandler* class intended to communicate with a remote machine whose address is given by *host* and *port*.

Alterado na versão 3.4: If `port` is specified as `None`, a Unix domain socket is created using the value in `host` - otherwise, a UDP socket is created.

emit()
Pickles the record's attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. To unpickle the record at the receiving end into a *LogRecord*, use the *makeLogRecord()* function.

makeSocket()
The factory method of *SocketHandler* is here overridden to create a UDP socket (*socket.SOCK_DGRAM*).

send(*s*)
Send a pickled byte-string to a socket. The format of the sent byte-string is as described in the documentation for *SocketHandler.makePickle()*.

16.8.10 SysLogHandler

The *SysLogHandler* class, located in the *logging.handlers* module, supports sending logging messages to a remote or local Unix syslog.

class logging.handlers.**SysLogHandler**(*address=('localhost', SYSLOG_UDP_PORT), facility=LOG_USER, socktype=socket.SOCK_DGRAM*)
Returns a new instance of the *SysLogHandler* class intended to communicate with a remote Unix machine whose address is given by *address* in the form of a `(host, port)` tuple. If *address* is not specified, `('localhost', 514)` is used. The address is used to open a socket. An alternative to providing a `(host, port)` tuple is providing an address as a string, for example '/dev/log'. In this case, a Unix domain socket is used to send the message to the syslog. If *facility* is not specified, `LOG_USER` is used. The type of socket opened depends on the *socktype* argument, which defaults to *socket.SOCK_DGRAM* and thus opens a UDP socket. To open a TCP socket (for use with the newer syslog daemons such as rsyslog), specify a value of *socket.SOCK_STREAM*.

Note that if your server is not listening on UDP port 514, *SysLogHandler* may appear not to work. In that case, check what address you should be using for a domain socket - it's system dependent. For example, on Linux it's usually '/dev/log' but on OS/X it's '/var/run/syslog'. You'll need to check your platform and use the appropriate address (you may need to do this check at runtime if your application needs to run on several platforms). On Windows, you pretty much have to use the UDP option.

Alterado na versão 3.2: *socktype* was added.

close()
Closes the socket to the remote host.

emit(*record*)
The record is formatted, and then sent to the syslog server. If exception information is present, it is *not* sent to the server.

Alterado na versão 3.2.1: (See: bpo-12168.) In earlier versions, the message sent to the syslog daemons was always terminated with a NUL byte, because early versions of these daemons expected a NUL terminated message - even though it's not in the relevant specification (**RFC 5424**). More recent versions of these daemons don't expect the NUL byte but strip it off if it's there, and even more recent daemons (which adhere more closely to RFC 5424) pass the NUL byte on as part of the message.

To enable easier handling of syslog messages in the face of all these differing daemon behaviours, the appending of the NUL byte has been made configurable, through the use of a class-level attribute, `append_nul`. This defaults to `True` (preserving the existing behaviour) but can be set to `False` on a `SysLogHandler` instance in order for that instance to *not* append the NUL terminator.

Alterado na versão 3.3: (See: bpo-12419.) In earlier versions, there was no facility for an "ident" or "tag" prefix to identify the source of the message. This can now be specified using a class-level attribute, defaulting to `""` to preserve existing behaviour, but which can be overridden on a `SysLogHandler` instance in order

for that instance to prepend the ident to every message handled. Note that the provided ident must be text, not bytes, and is prepended to the message exactly as is.

encodePriority(*facility*, *priority*)

Encodes the facility and priority into an integer. You can pass in strings or integers - if strings are passed, internal mapping dictionaries are used to convert them to integers.

The symbolic `LOG_` values are defined in *SysLogHandler* and mirror the values defined in the `sys/syslog.h` header file.

Priorities

Name (string)	Symbolic value
`alert`	LOG_ALERT
`crit` or `critical`	LOG_CRIT
`debug`	LOG_DEBUG
`emerg` or `panic`	LOG_EMERG
`err` ou `error`	LOG_ERR
`info`	LOG_INFO
`notice`	LOG_NOTICE
`warn` or `warning`	LOG_WARNING

Facilities

Name (string)	Symbolic value
`auth`	LOG_AUTH
`authpriv`	LOG_AUTHPRIV
`cron`	LOG_CRON
`daemon`	LOG_DAEMON
`ftp`	LOG_FTP
`kern`	LOG_KERN
`lpr`	LOG_LPR
`mail`	LOG_MAIL
`news`	LOG_NEWS
`syslog`	LOG_SYSLOG
`user`	LOG_USER
`uucp`	LOG_UUCP
`local0`	LOG_LOCAL0
`local1`	LOG_LOCAL1
`local2`	LOG_LOCAL2
`local3`	LOG_LOCAL3
`local4`	LOG_LOCAL4
`local5`	LOG_LOCAL5
`local6`	LOG_LOCAL6
`local7`	LOG_LOCAL7

mapPriority(*levelname*)

Maps a logging level name to a syslog priority name. You may need to override this if you are using custom levels, or if the default algorithm is not suitable for your needs. The default algorithm maps `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL` to the equivalent syslog names, and all other level names to 'warning'.

16.8.11 NTEventLogHandler

The *NTEventLogHandler* class, located in the *logging.handlers* module, supports sending logging messages to a local Windows NT, Windows 2000 or Windows XP event log. Before you can use it, you need Mark Hammond's Win32 extensions for Python installed.

class logging.handlers.**NTEventLogHandler**(*appname*, *dllname=None*, *logtype='Application'*)

Returns a new instance of the *NTEventLogHandler* class. The *appname* is used to define the application name as it appears in the event log. An appropriate registry entry is created using this name. The *dllname* should give the fully qualified pathname of a .dll or .exe which contains message definitions to hold in the log (if not specified, `'win32service.pyd'` is used - this is installed with the Win32 extensions and contains some basic placeholder message definitions. Note that use of these placeholders will make your event logs big, as the entire message source is held in the log. If you want slimmer logs, you have to pass in the name of your own .dll or .exe which contains the message definitions you want to use in the event log). The *logtype* is one of `'Application'`, `'System'` or `'Security'`, and defaults to `'Application'`.

close()

At this point, you can remove the application name from the registry as a source of event log entries. However, if you do this, you will not be able to see the events as you intended in the Event Log Viewer - it needs to be able to access the registry to get the .dll name. The current version does not do this.

emit(*record*)

Determines the message ID, event category and event type, and then logs the message in the NT event log.

getEventCategory(*record*)

Returns the event category for the record. Override this if you want to specify your own categories. This version returns 0.

getEventType(*record*)

Returns the event type for the record. Override this if you want to specify your own types. This version does a mapping using the handler's typemap attribute, which is set up in `__init__()` to a dictionary which contains mappings for `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL`. If you are using your own levels, you will either need to override this method or place a suitable dictionary in the handler's *typemap* attribute.

getMessageID(*record*)

Returns the message ID for the record. If you are using your own messages, you could do this by having the *msg* passed to the logger being an ID rather than a format string. Then, in here, you could use a dictionary lookup to get the message ID. This version returns 1, which is the base message ID in `win32service.pyd`.

16.8.12 SMTPHandler

The *SMTPHandler* class, located in the *logging.handlers* module, supports sending logging messages to an email address via SMTP.

class logging.handlers.**SMTPHandler**(*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*, *timeout=1.0*)

Returns a new instance of the *SMTPHandler* class. The instance is initialized with the from and to addresses and subject line of the email. The *toaddrs* should be a list of strings. To specify a non-standard SMTP port, use the (host, port) tuple format for the *mailhost* argument. If you use a string, the standard SMTP port is used. If your SMTP server requires authentication, you can specify a (username, password) tuple for the *credentials* argument.

To specify the use of a secure protocol (TLS), pass in a tuple to the *secure* argument. This will only be used when authentication credentials are supplied. The tuple should be either an empty tuple, or a single-value tuple with the name of a keyfile, or a 2-value tuple with the names of the keyfile and certificate file. (This tuple is passed to the *smtplib.SMTP.starttls()* method.)

A timeout can be specified for communication with the SMTP server using the *timeout* argument.

Novo na versão 3.3: The *timeout* argument was added.

emit(*record*)
Formats the record and sends it to the specified addressees.

getSubject(*record*)
If you want to specify a subject line which is record-dependent, override this method.

16.8.13 MemoryHandler

The *MemoryHandler* class, located in the *logging.handlers* module, supports buffering of logging records in memory, periodically flushing them to a *target* handler. Flushing occurs whenever the buffer is full, or when an event of a certain severity or greater is seen.

MemoryHandler is a subclass of the more general *BufferingHandler*, which is an abstract class. This buffers logging records in memory. Whenever each record is added to the buffer, a check is made by calling `shouldFlush()` to see if the buffer should be flushed. If it should, then `flush()` is expected to do the flushing.

class logging.handlers.**BufferingHandler**(*capacity*)
Initializes the handler with a buffer of the specified capacity. Here, *capacity* means the number of logging records buffered.

emit(*record*)
Append the record to the buffer. If *shouldFlush()* returns true, call *flush()* to process the buffer.

flush()
You can override this to implement custom flushing behavior. This version just zaps the buffer to empty.

shouldFlush(*record*)
Return `True` if the buffer is up to capacity. This method can be overridden to implement custom flushing strategies.

class logging.handlers.**MemoryHandler**(*capacity*, *flushLevel=ERROR*, *target=None*, *flushOnClose=True*)
Returns a new instance of the *MemoryHandler* class. The instance is initialized with a buffer size of *capacity* (number of records buffered). If *flushLevel* is not specified, `ERROR` is used. If no *target* is specified, the target will need to be set using *setTarget()* before this handler does anything useful. If *flushOnClose* is specified as `False`, then the buffer is *not* flushed when the handler is closed. If not specified or specified as `True`, the previous behaviour of flushing the buffer will occur when the handler is closed.

Alterado na versão 3.6: The *flushOnClose* parameter was added.

close()
Calls *flush()*, sets the target to `None` and clears the buffer.

flush()
For a *MemoryHandler*, flushing means just sending the buffered records to the target, if there is one. The buffer is also cleared when this happens. Override if you want different behavior.

setTarget(*target*)
Sets the target handler for this handler.

shouldFlush(*record*)
Checks for buffer full or a record at the *flushLevel* or higher.

16.8.14 HTTPHandler

The *HTTPHandler* class, located in the *logging.handlers* module, supports sending logging messages to a Web server, using either GET or POST semantics.

class logging.handlers.**HTTPHandler**(*host*, *url*, *method='GET'*, *secure=False*, *credentials=None*, *context=None*)

Returns a new instance of the *HTTPHandler* class. The *host* can be of the form host:port, should you need to use a specific port number. If no *method* is specified, GET is used. If *secure* is true, a HTTPS connection will be used. The *context* parameter may be set to a *ssl.SSLContext* instance to configure the SSL settings used for the HTTPS connection. If *credentials* is specified, it should be a 2-tuple consisting of userid and password, which will be placed in a HTTP 'Authorization' header using Basic authentication. If you specify credentials, you should also specify secure=True so that your userid and password are not passed in cleartext across the wire.

Alterado na versão 3.5: The *context* parameter was added.

mapLogRecord(*record*)

Provides a dictionary, based on record, which is to be URL-encoded and sent to the web server. The default implementation just returns record.__dict__. This method can be overridden if e.g. only a subset of *LogRecord* is to be sent to the web server, or if more specific customization of what's sent to the server is required.

emit(*record*)

Sends the record to the Web server as a URL-encoded dictionary. The *mapLogRecord()* method is used to convert the record to the dictionary to be sent.

Nota: Since preparing a record for sending it to a Web server is not the same as a generic formatting operation, using *setFormatter()* to specify a *Formatter* for a *HTTPHandler* has no effect. Instead of calling *format()*, this handler calls *mapLogRecord()* and then *urllib.parse.urlencode()* to encode the dictionary in a form suitable for sending to a Web server.

16.8.15 QueueHandler

Novo na versão 3.2.

The *QueueHandler* class, located in the *logging.handlers* module, supports sending logging messages to a queue, such as those implemented in the *queue* or *multiprocessing* modules.

Along with the *QueueListener* class, *QueueHandler* can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in Web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via *SMTPHandler*) are done on a separate thread.

class logging.handlers.**QueueHandler**(*queue*)

Returns a new instance of the *QueueHandler* class. The instance is initialized with the queue to send messages to. The *queue* can be any queue-like object; it's used as-is by the *enqueue()* method, which needs to know how to send messages to it. The queue is not *required* to have the task tracking API, which means that you can use *SimpleQueue* instances for *queue*.

emit(*record*)

Enqueues the result of preparing the LogRecord. Should an exception occur (e.g. because a bounded queue has filled up), the *handleError()* method is called to handle the error. This can result in the record silently being dropped (if logging.raiseExceptions is False) or a message printed to sys.stderr (if logging.raiseExceptions is True).

prepare(*record*)
Prepares a record for queuing. The object returned by this method is enqueued.

The base implementation formats the record to merge the message, arguments, and exception information, if present. It also removes unpickleable items from the record in-place.

You might want to override this method if you want to convert the record to a dict or JSON string, or send a modified copy of the record while leaving the original intact.

enqueue(*record*)
Enqueues the record on the queue using `put_nowait()`; you may want to override this if you want to use blocking behaviour, or a timeout, or a customized queue implementation.

16.8.16 QueueListener

Novo na versão 3.2.

The *QueueListener* class, located in the *logging.handlers* module, supports receiving logging messages from a queue, such as those implemented in the *queue* or *multiprocessing* modules. The messages are received from a queue in an internal thread and passed, on the same thread, to one or more handlers for processing. While *QueueListener* is not itself a handler, it is documented here because it works hand-in-hand with *QueueHandler*.

Along with the *QueueHandler* class, *QueueListener* can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in Web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via *SMTPHandler*) are done on a separate thread.

class logging.handlers.**QueueListener**(*queue*, **handlers*, *respect_handler_level=False*)
Returns a new instance of the *QueueListener* class. The instance is initialized with the queue to send messages to and a list of handlers which will handle entries placed on the queue. The queue can be any queue-like object; it's passed as-is to the *dequeue()* method, which needs to know how to get messages from it. The queue is not *required* to have the task tracking API (though it's used if available), which means that you can use *SimpleQueue* instances for *queue*.

If `respect_handler_level` is `True`, a handler's level is respected (compared with the level for the message) when deciding whether to pass messages to that handler; otherwise, the behaviour is as in previous Python versions - to always pass each message to each handler.

Alterado na versão 3.5: The `respect_handler_level` argument was added.

dequeue(*block*)
Dequeues a record and return it, optionally blocking.

The base implementation uses `get()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

prepare(*record*)
Prepare a record for handling.

This implementation just returns the passed-in record. You may want to override this method if you need to do any custom marshalling or manipulation of the record before passing it to the handlers.

handle(*record*)
Handle a record.

This just loops through the handlers offering them the record to handle. The actual object passed to the handlers is that which is returned from *prepare()*.

start()
Starts the listener.

This starts up a background thread to monitor the queue for LogRecords to process.

stop()
Stops the listener.

This asks the thread to terminate, and then waits for it to do so. Note that if you don't call this before your application exits, there may be some records still left on the queue, which won't be processed.

enqueue_sentinel()
Writes a sentinel to the queue to tell the listener to quit. This implementation uses `put_nowait()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

Novo na versão 3.3.

Ver também:

Módulo *logging* Referência da API para o módulo de logging.

Módulo *logging.config* API de configuração para o módulo logging.

16.9 getpass — Entrada de senha portátil

Código Fonte: Lib/getpass.py

O módulo *getpass* fornece duas funções:

getpass.**getpass**(*prompt='Password: '*, *stream=None*)
Solicita uma senha do usuário sem emiti-la. O usuário é solicitado usando a string *prompt*, cujo padrão é `'Password: '`. No Unix, o prompt é escrito no objeto arquivo ou similar *stream* usando o tratador de erros de substituição, se necessário. O *stream* padrão para o terminal de controle (`/dev/tty`) ou se não estiver disponível para `sys.stderr` (este argumento é ignorado no Windows).

Se uma entrada sem exibição em tela não estiver disponível, getpass() recorre a exibir uma mensagem de aviso para *stream* e lê de `sys.stdin` e levantar de um *GetPassWarning*.

Nota: Se você chamar getpass de dentro do IDLE, a entrada pode ser feita no terminal de onde você iniciou o IDLE, e não na própria janela ociosa.

exception getpass.**GetPassWarning**
A subclasse *UserWarning* é levantada quando a entrada de senha pode acabar sendo exibida na tela.

getpass.**getuser**()
Retorna o "nome de login" do usuário.

Esta função verifica as variáveis de ambiente `LOGNAME`, `USER`, `LNAME` e `USERNAME`, nesta ordem, e retorna o valor da primeiro que estiver definida como uma string não vazia. Se nenhuma estiver definida, o nome de login do banco de dados de senhas é retornado em sistemas que suportam o módulo *pwd*, caso contrário, uma exceção é levantada.

Em geral, esta função deve ter preferência sobre *os.getlogin()*.

16.10 curses — Gerenciador de terminal para visualizadores de células de caracteres.

O módulo *curses* provê uma interface para a livraria curses, o padrão de-facto para manuseio avançado de terminal portável.

While curses is most widely used in the Unix environment, versions are available for Windows, DOS, and possibly other systems as well. This extension module is designed to match the API of ncurses, an open-source curses library hosted on Linux and the BSD variants of Unix.

Nota: Whenever the documentation mentions a *character* it can be specified as an integer, a one-character Unicode string or a one-byte byte string.

Whenever the documentation mentions a *character string* it can be specified as a Unicode string or a byte string.

Nota: Desde a versão 5.4, a livraria ncurses decidiu como interpretar dados não-ASCII usando a função `nl_langinfo`. Isso significa que você terá que chamar a função *locale.setlocale()* na aplicação e codificar strings Unicode usando um dos codificadores disponíveis por padrão no sistema. Esse exemplo usa o codificador padrão do sistema:

```
import locale
locale.setlocale(locale.LC_ALL, '')
code = locale.getpreferredencoding()
```

Então faça uso de *code* como codificador para as chamadas *str.encode()*.

Ver também:

Módulo *curses.ascii* Utilidades para trabalhar com caracteres ASCII, independentemente de suas configurações locais.

Módulo *curses.panel* Uma extensão de painel stackeada que adiciona profundidade às janelas do curses.

Módulo *curses.textpad* Widget de texto editável para suporte ao curses **Emacs**-like bindings.

curses-howto Tutorial material on using curses with Python, by Andrew Kuchling and Eric Raymond.

The Tools/demo/ directory in the Python source distribution contains some example programs using the curses bindings provided by this module.

16.10.1 Funções

The module *curses* defines the following exception:

exception curses.**error**
Exception raised when a curses library function returns an error.

Nota: Whenever *x* or *y* arguments to a function or a method are optional, they default to the current cursor location. Whenever *attr* is optional, it defaults to `A_NORMAL`.

The module *curses* defines the following functions:

curses.**baudrate**()
Return the output speed of the terminal in bits per second. On software terminal emulators it will have a fixed high value. Included for historical reasons; in former times, it was used to write output loops for time delays and occasionally to change interfaces depending on the line speed.

curses.**beep**()
Emit a short attention sound.

curses.**can_change_color**()
Return `True` or `False`, depending on whether the programmer can change the colors displayed by the terminal.

curses.**cbreak**()
Enter cbreak mode. In cbreak mode (sometimes called "rare" mode) normal tty line buffering is turned off and characters are available to be read one by one. However, unlike raw mode, special characters (interrupt, quit, suspend, and flow control) retain their effects on the tty driver and calling program. Calling first *raw()* then *cbreak()* leaves the terminal in cbreak mode.

curses.**color_content**(*color_number*)
Return the intensity of the red, green, and blue (RGB) components in the color *color_number*, which must be between `0` and `COLORS`. Return a 3-tuple, containing the R,G,B values for the given color, which will be between `0` (no component) and `1000` (maximum amount of component).

curses.**color_pair**(*color_number*)
Return the attribute value for displaying text in the specified color. This attribute value can be combined with `A_STANDOUT`, `A_REVERSE`, and the other `A_*` attributes. *pair_number()* is the counterpart to this function.

curses.**curs_set**(*visibility*)
Set the cursor state. *visibility* can be set to `0`, `1`, or `2`, for invisible, normal, or very visible. If the terminal supports the visibility requested, return the previous cursor state; otherwise raise an exception. On many terminals, the "visible" mode is an underline cursor and the "very visible" mode is a block cursor.

curses.**def_prog_mode**()
Save the current terminal mode as the "program" mode, the mode when the running program is using curses. (Its counterpart is the "shell" mode, for when the program is not in curses.) Subsequent calls to *reset_prog_mode()* will restore this mode.

curses.**def_shell_mode**()
Save the current terminal mode as the "shell" mode, the mode when the running program is not using curses. (Its counterpart is the "program" mode, when the program is using curses capabilities.) Subsequent calls to *reset_shell_mode()* will restore this mode.

curses.**delay_output**(*ms*)
Insert an *ms* millisecond pause in output.

curses.**doupdate**()
Atualiza a tela física. A biblioteca do curses mantém duas estruturas de dados, uma representando o conteúdo da tela física atual e uma tela virtual representando o próximo estado desejado. O *doupdate()* atualiza a tela física para corresponder à tela virtual.

The virtual screen may be updated by a *noutrefresh()* call after write operations such as *addstr()* have been performed on a window. The normal *refresh()* call is simply `noutrefresh()` followed by `doupdate()`; if you have to update multiple windows, you can speed performance and perhaps reduce screen flicker by issuing `noutrefresh()` calls on all windows, followed by a single `doupdate()`.

curses.**echo**()
Enter echo mode. In echo mode, each character input is echoed to the screen as it is entered.

curses.**endwin**()
De-initialize the library, and return terminal to normal status.

curses.**erasechar**()
Return the user's current erase character as a one-byte bytes object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.**filter**()
The *filter()* routine, if used, must be called before *initscr()* is called. The effect is that, during those calls, LINES is set to 1; the capabilities clear, cup, cud, cud1, cuu1, cuu, vpa are disabled; and the home string is set to the value of cr. The effect is that the cursor is confined to the current line, and so are screen updates. This may be used for enabling character-at-a-time line editing without touching the rest of the screen.

curses.**flash**()
Flash the screen. That is, change it to reverse-video and then change it back in a short interval. Some people prefer such as 'visible bell' to the audible attention signal produced by *beep()*.

curses.**flushinp**()
Flush all input buffers. This throws away any typeahead that has been typed by the user and has not yet been processed by the program.

curses.**getmouse**()
After *getch()* returns KEY_MOUSE to signal a mouse event, this method should be call to retrieve the queued mouse event, represented as a 5-tuple (id, x, y, z, bstate). *id* is an ID value used to distinguish multiple devices, and *x*, *y*, *z* are the event's coordinates. (*z* is currently unused.) *bstate* is an integer value whose bits will be set to indicate the type of event, and will be the bitwise OR of one or more of the following constants, where *n* is the button number from 1 to 4: BUTTONn_PRESSED, BUTTONn_RELEASED, BUTTONn_CLICKED, BUTTONn_DOUBLE_CLICKED, BUTTONn_TRIPLE_CLICKED, BUTTON_SHIFT, BUTTON_CTRL, BUTTON_ALT.

curses.**getsyx**()
Return the current coordinates of the virtual screen cursor as a tuple (y, x). If *leaveok* is currently True, then return (-1, -1).

curses.**getwin**(*file*)
Read window related data stored in the file by an earlier putwin() call. The routine then creates and initializes a new window using that data, returning the new window object.

curses.**has_colors**()
Return True if the terminal can display colors; otherwise, return False.

curses.**has_ic**()
Return True if the terminal has insert- and delete-character capabilities. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

curses.**has_il**()
Return True if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

curses.**has_key**(*ch*)
Take a key value *ch*, and return True if the current terminal type recognizes a key with that value.

curses.**halfdelay**(*tenths*)
Used for half-delay mode, which is similar to cbreak mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, raise an exception if nothing has been typed. The value of *tenths* must be a number between 1 and 255. Use *nocbreak()* to leave half-delay mode.

curses.**init_color**(*color_number*, *r*, *g*, *b*)
Change the definition of a color, taking the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of *color_number* must be between 0 and COLORS. Each of *r*, *g*, *b*, must be a value between 0 and 1000. When *init_color()* is used, all occurrences of that

color on the screen immediately change to the new definition. This function is a no-op on most terminals; it is active only if *can_change_color()* returns `True`.

`curses.`**`init_pair`**(*pair_number*, *fg*, *bg*)
Change the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of *pair_number* must be between `1` and `COLOR_PAIRS - 1` (the `0` color pair is wired to white on black and cannot be changed). The value of *fg* and *bg* arguments must be between `0` and `COLORS`. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair are changed to the new definition.

`curses.`**`initscr`**()
Initialize the library. Return a *window* object which represents the whole screen.

Nota: If there is an error opening the terminal, the underlying curses library may cause the interpreter to exit.

`curses.`**`is_term_resized`**(*nlines*, *ncols*)
Return `True` if *resize_term()* would modify the window structure, `False` otherwise.

`curses.`**`isendwin`**()
Return `True` if *endwin()* has been called (that is, the curses library has been deinitialized).

`curses.`**`keyname`**(*k*)
Return the name of the key numbered *k* as a bytes object. The name of a key generating printable ASCII character is the key's character. The name of a control-key combination is a two-byte bytes object consisting of a caret (`b'^'`) followed by the corresponding printable ASCII character. The name of an alt-key combination (128–255) is a bytes object consisting of the prefix `b'M-'` followed by the name of the corresponding ASCII character.

`curses.`**`killchar`**()
Return the user's current line kill character as a one-byte bytes object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

`curses.`**`longname`**()
Return a bytes object containing the terminfo long name field describing the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to *initscr()*.

`curses.`**`meta`**(*flag*)
If *flag* is `True`, allow 8-bit characters to be input. If *flag* is `False`, allow only 7-bit chars.

`curses.`**`mouseinterval`**(*interval*)
Set the maximum time in milliseconds that can elapse between press and release events in order for them to be recognized as a click, and return the previous interval value. The default value is 200 msec, or one fifth of a second.

`curses.`**`mousemask`**(*mousemask*)
Set the mouse events to be reported, and return a tuple `(availmask, oldmask)`. *availmask* indicates which of the specified mouse events can be reported; on complete failure it returns `0`. *oldmask* is the previous value of the given window's mouse event mask. If this function is never called, no mouse events are ever reported.

`curses.`**`napms`**(*ms*)
Sleep for *ms* milliseconds.

`curses.`**`newpad`**(*nlines*, *ncols*)
Create and return a pointer to a new pad data structure with the given number of lines and columns. Return a pad as a window object.

A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (such as from scrolling or echoing of input) do not occur. The *refresh()* and *noutrefresh()* methods of a pad require 6 arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display. The arguments are *pminrow*,

pmincol, *sminrow*, *smincol*, *smaxrow*, *smaxcol*; the *p* arguments refer to the upper left corner of the pad region to be displayed and the *s* arguments define a clipping box on the screen within which the pad region is to be displayed.

curses.**newwin**(*nlines*, *ncols*)
curses.**newwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)
Return a new *window*, whose left-upper corner is at `(begin_y, begin_x)`, and whose height/width is *nlines*/*ncols*.

By default, the window will extend from the specified position to the lower right corner of the screen.

curses.**nl**()
Enter newline mode. This mode translates the return key into newline on input, and translates newline into return and line-feed on output. Newline mode is initially on.

curses.**nocbreak**()
Leave cbreak mode. Return to normal "cooked" mode with line buffering.

curses.**noecho**()
Leave echo mode. Echoing of input characters is turned off.

curses.**nonl**()
Leave newline mode. Disable translation of return into newline on input, and disable low-level translation of newline into newline/return on output (but this does not change the behavior of `addch('\n')`, which always does the equivalent of return and line feed on the virtual screen). With translation off, curses can sometimes speed up vertical motion a little; also, it will be able to detect the return key on input.

curses.**noqiflush**()
When the `noqiflush()` routine is used, normal flush of input and output queues associated with the `INTR`, `QUIT` and `SUSP` characters will not be done. You may want to call `noqiflush()` in a signal handler if you want output to continue as though the interrupt had not occurred, after the handler exits.

curses.**noraw**()
Leave raw mode. Return to normal "cooked" mode with line buffering.

curses.**pair_content**(*pair_number*)
Return a tuple `(fg, bg)` containing the colors for the requested color pair. The value of *pair_number* must be between `1` and `COLOR_PAIRS - 1`.

curses.**pair_number**(*attr*)
Return the number of the color-pair set by the attribute value *attr*. *color_pair()* is the counterpart to this function.

curses.**putp**(*str*)
Equivalent to `tputs(str, 1, putchar)`; emit the value of a specified terminfo capability for the current terminal. Note that the output of *putp()* always goes to standard output.

curses.**qiflush**([*flag*])
If *flag* is `False`, the effect is the same as calling *noqiflush()*. If *flag* is `True`, or no argument is provided, the queues will be flushed when these control characters are read.

curses.**raw**()
Enter raw mode. In raw mode, normal line buffering and processing of interrupt, quit, suspend, and flow control keys are turned off; characters are presented to curses input functions one by one.

curses.**reset_prog_mode**()
Restore the terminal to "program" mode, as previously saved by *def_prog_mode()*.

curses.**reset_shell_mode**()
Restore the terminal to "shell" mode, as previously saved by *def_shell_mode()*.

curses.**resetty**()
Restore the state of the terminal modes to what it was at the last call to *savetty()*.

curses.**resize_term**(*nlines*, *ncols*)
Backend function used by *resizeterm()*, performing most of the work; when resizing the windows, *resize_term()* blank-fills the areas that are extended. The calling application should fill in these areas with appropriate data. The resize_term() function attempts to resize all windows. However, due to the calling convention of pads, it is not possible to resize these without additional interaction with the application.

curses.**resizeterm**(*nlines*, *ncols*)
Resize the standard and current windows to the specified dimensions, and adjusts other bookkeeping data used by the curses library that record the window dimensions (in particular the SIGWINCH handler).

curses.**savetty**()
Save the current state of the terminal modes in a buffer, usable by *resetty()*.

curses.**setsyx**(*y*, *x*)
Set the virtual screen cursor to *y*, *x*. If *y* and *x* are both -1, then *leaveok* is set True.

curses.**setupterm**(*term=None*, *fd=-1*)
Initialize the terminal. *term* is a string giving the terminal name, or None; if omitted or None, the value of the TERM environment variable will be used. *fd* is the file descriptor to which any initialization sequences will be sent; if not supplied or -1, the file descriptor for sys.stdout will be used.

curses.**start_color**()
Must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after *initscr()*.

start_color() initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables in the *curses* module, COLORS and COLOR_PAIRS, containing the maximum number of colors and color-pairs the terminal can support. It also restores the colors on the terminal to the values they had when the terminal was just turned on.

curses.**termattrs**()
Return a logical OR of all video attributes supported by the terminal. This information is useful when a curses program needs complete control over the appearance of the screen.

curses.**termname**()
Return the value of the environment variable TERM, as a bytes object, truncated to 14 characters.

curses.**tigetflag**(*capname*)
Return the value of the Boolean capability corresponding to the terminfo capability name *capname* as an integer. Return the value -1 if *capname* is not a Boolean capability, or 0 if it is canceled or absent from the terminal description.

curses.**tigetnum**(*capname*)
Return the value of the numeric capability corresponding to the terminfo capability name *capname* as an integer. Return the value -2 if *capname* is not a numeric capability, or -1 if it is canceled or absent from the terminal description.

curses.**tigetstr**(*capname*)
Return the value of the string capability corresponding to the terminfo capability name *capname* as a bytes object. Return None if *capname* is not a terminfo "string capability", or is canceled or absent from the terminal description.

curses.**tparm**(*str*[, ...])
Instantiate the bytes object *str* with the supplied parameters, where *str* should be a parameterized string obtained from the terminfo database. E.g. tparm(tigetstr("cup"), 5, 3) could result in b'\033[6;4H', the exact result depending on terminal type.

curses.**typeahead**(*fd*)
Specify that the file descriptor *fd* be used for typeahead checking. If *fd* is -1, then no typeahead checking is done.

The curses library does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until refresh or doupdate is

called again, allowing faster response to commands typed in advance. This function allows specifying a different file descriptor for typeahead checking.

curses.**unctrl**(*ch*)
Return a bytes object which is a printable representation of the character *ch*. Control characters are represented as a caret followed by the character, for example as `b'^C'`. Printing characters are left as they are.

curses.**ungetch**(*ch*)
Push *ch* so the next *getch()* will return it.

Nota: Only one *ch* can be pushed before `getch()` is called.

curses.**update_lines_cols**()
Update `LINES` and `COLS`. Useful for detecting manual screen resize.

Novo na versão 3.5.

curses.**unget_wch**(*ch*)
Push *ch* so the next *get_wch()* will return it.

Nota: Only one *ch* can be pushed before `get_wch()` is called.

Novo na versão 3.3.

curses.**ungetmouse**(*id*, *x*, *y*, *z*, *bstate*)
Push a `KEY_MOUSE` event onto the input queue, associating the given state data with it.

curses.**use_env**(*flag*)
If used, this function should be called before *initscr()* or newterm are called. When *flag* is `False`, the values of lines and columns specified in the terminfo database will be used, even if environment variables `LINES` and `COLUMNS` (used by default) are set, or if curses is running in a window (in which case default behavior would be to use the window size if `LINES` and `COLUMNS` are not set).

curses.**use_default_colors**()
Allow use of default values for colors on terminals supporting this feature. Use this to support transparency in your application. The default color is assigned to the color number `-1`. After calling this function, `init_pair(x, curses.COLOR_RED, -1)` initializes, for instance, color pair *x* to a red foreground color on the default background.

curses.**wrapper**(*func*, ...)
Initialize curses and call another callable object, *func*, which should be the rest of your curses-using application. If the application raises an exception, this function will restore the terminal to a sane state before re-raising the exception and generating a traceback. The callable object *func* is then passed the main window 'stdscr' as its first argument, followed by any other arguments passed to `wrapper()`. Before calling *func*, `wrapper()` turns on cbreak mode, turns off echo, enables the terminal keypad, and initializes colors if the terminal has color support. On exit (whether normally or by exception) it restores cooked mode, turns on echo, and disables the terminal keypad.

16.10.2 Window Objects

Window objects, as returned by `initscr()` and `newwin()` above, have the following methods and attributes:

`window.`**`addch`**(*ch*[, *attr*])
`window.`**`addch`**(*y*, *x*, *ch*[, *attr*])
Paint character *ch* at `(y, x)` with attributes *attr*, overwriting any character previously painter at that location. By default, the character position and attributes are the current settings for the window object.

Nota: Writing outside the window, subwindow, or pad raises a `curses.error`. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the character is printed.

`window.`**`addnstr`**(*str*, *n*[, *attr*])
`window.`**`addnstr`**(*y*, *x*, *str*, *n*[, *attr*])
Paint at most *n* characters of the character string *str* at `(y, x)` with attributes *attr*, overwriting anything previously on the display.

`window.`**`addstr`**(*str*[, *attr*])
`window.`**`addstr`**(*y*, *x*, *str*[, *attr*])
Paint the character string *str* at `(y, x)` with attributes *attr*, overwriting anything previously on the display.

Nota: Writing outside the window, subwindow, or pad raises `curses.error`. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the string is printed.

`window.`**`attroff`**(*attr*)
Remove attribute *attr* from the "background" set applied to all writes to the current window.

`window.`**`attron`**(*attr*)
Add attribute *attr* from the "background" set applied to all writes to the current window.

`window.`**`attrset`**(*attr*)
Set the "background" set of attributes to *attr*. This set is initially `0` (no attributes).

`window.`**`bkgd`**(*ch*[, *attr*])
Set the background property of the window to the character *ch*, with attributes *attr*. The change is then applied to every character position in that window:

- The attribute of every character in the window is changed to the new background attribute.
- Wherever the former background character appears, it is changed to the new background character.

`window.`**`bkgdset`**(*ch*[, *attr*])
Set the window's background. A window's background consists of a character and any combination of attributes. The attribute part of the background is combined (OR'ed) with all non-blank characters that are written into the window. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations.

`window.`**`border`**([*ls*[, *rs*[, *ts*[, *bs*[, *tl*[, *tr*[, *bl*[, *br*]]]]]]]])
Draw a border around the edges of the window. Each parameter specifies the character to use for a specific part of the border; see the table below for more details.

Nota: A `0` value for any parameter will cause the default character to be used for that parameter. Keyword parameters can *not* be used. The defaults are listed in this table:

Parameter	Description (descrição)	Valor padrão
ls	Left side	`ACS_VLINE`
rs	Right side	`ACS_VLINE`
ts	Top	`ACS_HLINE`
bs	Bottom	`ACS_HLINE`
tl	Upper-left corner	`ACS_ULCORNER`
tr	Upper-right corner	`ACS_URCORNER`
bl	Bottom-left corner	`ACS_LLCORNER`
br	Bottom-right corner	`ACS_LRCORNER`

`window.`**`box`**`(`[*vertch*, *horch*]`)`
Similar to `border()`, but both *ls* and *rs* are *vertch* and both *ts* and *bs* are *horch*. The default corner characters are always used by this function.

`window.`**`chgat`**`(`*attr*`)`
`window.`**`chgat`**`(`*num*, *attr*`)`
`window.`**`chgat`**`(`*y*, *x*, *attr*`)`
`window.`**`chgat`**`(`*y*, *x*, *num*, *attr*`)`
Set the attributes of *num* characters at the current cursor position, or at position `(y, x)` if supplied. If *num* is not given or is `-1`, the attribute will be set on all the characters to the end of the line. This function moves cursor to position `(y, x)` if supplied. The changed line will be touched using the `touchline()` method so that the contents will be redisplayed by the next window refresh.

`window.`**`clear`**`()`
Like `erase()`, but also cause the whole window to be repainted upon next call to `refresh()`.

`window.`**`clearok`**`(`*flag*`)`
If *flag* is `True`, the next call to `refresh()` will clear the window completely.

`window.`**`clrtobot`**`()`
Erase from cursor to the end of the window: all lines below the cursor are deleted, and then the equivalent of `clrtoeol()` is performed.

`window.`**`clrtoeol`**`()`
Erase from cursor to the end of the line.

`window.`**`cursyncup`**`()`
Update the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

`window.`**`delch`**`(`[*y*, *x*]`)`
Delete any character at `(y, x)`.

`window.`**`deleteln`**`()`
Delete the line under the cursor. All following lines are moved up by one line.

`window.`**`derwin`**`(`*begin_y*, *begin_x*`)`
`window.`**`derwin`**`(`*nlines*, *ncols*, *begin_y*, *begin_x*`)`
An abbreviation for "derive window", `derwin()` is the same as calling `subwin()`, except that *begin_y* and *begin_x* are relative to the origin of the window, rather than relative to the entire screen. Return a window object for the derived window.

`window.`**`echochar`**`(`*ch*[, *attr*]`)`
Add character *ch* with attribute *attr*, and immediately call `refresh()` on the window.

`window.`**`enclose`**`(`*y*, *x*`)`
Test whether the given pair of screen-relative character-cell coordinates are enclosed by the given window, returning

`True` or `False`. It is useful for determining what subset of the screen windows enclose the location of a mouse event.

`window.`**`encoding`**
Encoding used to encode method arguments (Unicode strings and characters). The encoding attribute is inherited from the parent window when a subwindow is created, for example with *`window.subwin()`*. By default, the locale encoding is used (see *`locale.getpreferredencoding()`*).

Novo na versão 3.3.

`window.`**`erase`**`()`
Clear the window.

`window.`**`getbegyx`**`()`
Return a tuple `(y, x)` of co-ordinates of upper-left corner.

`window.`**`getbkgd`**`()`
Return the given window's current background character/attribute pair.

`window.`**`getch`**`(`[*y*, *x*]`)`
Get a character. Note that the integer returned does *not* have to be in ASCII range: function keys, keypad keys and so on are represented by numbers higher than 255. In no-delay mode, return `-1` if there is no input, otherwise wait until a key is pressed.

`window.`**`get_wch`**`(`[*y*, *x*]`)`
Get a wide character. Return a character for most keys, or an integer for function keys, keypad keys, and other special keys. In no-delay mode, raise an exception if there is no input.

Novo na versão 3.3.

`window.`**`getkey`**`(`[*y*, *x*]`)`
Get a character, returning a string instead of an integer, as *`getch()`* does. Function keys, keypad keys and other special keys return a multibyte string containing the key name. In no-delay mode, raise an exception if there is no input.

`window.`**`getmaxyx`**`()`
Return a tuple `(y, x)` of the height and width of the window.

`window.`**`getparyx`**`()`
Return the beginning coordinates of this window relative to its parent window as a tuple `(y, x)`. Return `(-1, -1)` if this window has no parent.

`window.`**`getstr`**`()`
`window.`**`getstr`**`(`*n*`)`
`window.`**`getstr`**`(`*y*, *x*`)`
`window.`**`getstr`**`(`*y*, *x*, *n*`)`
Read a bytes object from the user, with primitive line editing capacity.

`window.`**`getyx`**`()`
Return a tuple `(y, x)` of current cursor position relative to the window's upper-left corner.

`window.`**`hline`**`(`*ch*, *n*`)`
`window.`**`hline`**`(`*y*, *x*, *ch*, *n*`)`
Display a horizontal line starting at `(y, x)` with length *n* consisting of the character *ch*.

`window.`**`idcok`**`(`*flag*`)`
If *flag* is `False`, curses no longer considers using the hardware insert/delete character feature of the terminal; if *flag* is `True`, use of character insertion and deletion is enabled. When curses is first initialized, use of character insert/delete is enabled by default.

window.**idlok**(*flag*)
If *flag* is `True`, *curses* will try and use hardware line editing facilities. Otherwise, line insertion/deletion are disabled.

window.**immedok**(*flag*)
If *flag* is `True`, any change in the window image automatically causes the window to be refreshed; you no longer have to call *refresh()* yourself. However, it may degrade performance considerably, due to repeated calls to wrefresh. This option is disabled by default.

window.**inch**([*y*, *x*])
Return the character at the given position in the window. The bottom 8 bits are the character proper, and upper bits are the attributes.

window.**insch**(*ch*[, *attr*])
window.**insch**(*y*, *x*, *ch*[, *attr*])
Paint character *ch* at `(y, x)` with attributes *attr*, moving the line from position *x* right by one character.

window.**insdelln**(*nlines*)
Insert *nlines* lines into the specified window above the current line. The *nlines* bottom lines are lost. For negative *nlines*, delete *nlines* lines starting with the one under the cursor, and move the remaining lines up. The bottom *nlines* lines are cleared. The current cursor position remains the same.

window.**insertln**()
Insert a blank line under the cursor. All following lines are moved down by one line.

window.**insnstr**(*str*, *n*[, *attr*])
window.**insnstr**(*y*, *x*, *str*, *n*[, *attr*])
Insert a character string (as many characters as will fit on the line) before the character under the cursor, up to *n* characters. If *n* is zero or negative, the entire string is inserted. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

window.**insstr**(*str*[, *attr*])
window.**insstr**(*y*, *x*, *str*[, *attr*])
Insert a character string (as many characters as will fit on the line) before the character under the cursor. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

window.**instr**([*n*])
window.**instr**(*y*, *x*[, *n*])
Return a bytes object of characters, extracted from the window starting at the current cursor position, or at *y*, *x* if specified. Attributes are stripped from the characters. If *n* is specified, *instr()* returns a string at most *n* characters long (exclusive of the trailing NUL).

window.**is_linetouched**(*line*)
Return `True` if the specified line was modified since the last call to *refresh()*; otherwise return `False`. Raise a *curses.error* exception if *line* is not valid for the given window.

window.**is_wintouched**()
Return `True` if the specified window was modified since the last call to *refresh()*; otherwise return `False`.

window.**keypad**(*flag*)
If *flag* is `True`, escape sequences generated by some keys (keypad, function keys) will be interpreted by *curses*. If *flag* is `False`, escape sequences will be left as is in the input stream.

window.**leaveok**(*flag*)
If *flag* is `True`, cursor is left where it is on update, instead of being at "cursor position." This reduces cursor movement where possible. If possible the cursor will be made invisible.

If *flag* is `False`, cursor will always be at "cursor position" after an update.

window.**move**(*new_y*, *new_x*)
Move cursor to `(new_y, new_x)`.

window.**mvderwin**(*y*, *x*)
Move the window inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

window.**mvwin**(*new_y*, *new_x*)
Move the window so its upper-left corner is at `(new_y, new_x)`.

window.**nodelay**(*flag*)
If *flag* is `True`, *getch()* will be non-blocking.

window.**notimeout**(*flag*)
If *flag* is `True`, escape sequences will not be timed out.

If *flag* is `False`, after a few milliseconds, an escape sequence will not be interpreted, and will be left in the input stream as is.

window.**noutrefresh**()
Mark for refresh but wait. This function updates the data structure representing the desired state of the window, but does not force an update of the physical screen. To accomplish that, call *doupdate()*.

window.**overlay**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])
Overlay the window on top of *destwin*. The windows need not be the same size, only the overlapping region is copied. This copy is non-destructive, which means that the current background character does not overwrite the old contents of *destwin*.

To get fine-grained control over the copied region, the second form of *overlay()* can be used. *sminrow* and *smincol* are the upper-left coordinates of the source window, and the other variables mark a rectangle in the destination window.

window.**overwrite**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])
Overwrite the window on top of *destwin*. The windows need not be the same size, in which case only the overlapping region is copied. This copy is destructive, which means that the current background character overwrites the old contents of *destwin*.

To get fine-grained control over the copied region, the second form of *overwrite()* can be used. *sminrow* and *smincol* are the upper-left coordinates of the source window, the other variables mark a rectangle in the destination window.

window.**putwin**(*file*)
Write all data associated with the window into the provided file object. This information can be later retrieved using the *getwin()* function.

window.**redrawln**(*beg*, *num*)
Indicate that the *num* screen lines, starting at line *beg*, are corrupted and should be completely redrawn on the next *refresh()* call.

window.**redrawwin**()
Touch the entire window, causing it to be completely redrawn on the next *refresh()* call.

window.**refresh**([*pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*])
Update the display immediately (sync actual screen with previous drawing/deleting methods).

The 6 optional arguments can only be specified when the window is a pad created with *newpad()*. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left-hand corner of the rectangle to be displayed in the pad. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles

must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow*, or *smincol* are treated as if they were zero.

window.**resize**(*nlines*, *ncols*)
Reallocate storage for a curses window to adjust its dimensions to the specified values. If either dimension is larger than the current values, the window's data is filled with blanks that have the current background rendition (as set by *bkgdset()*) merged into them.

window.**scroll**([*lines=1*])
Scroll the screen or scrolling region upward by *lines* lines.

window.**scrollok**(*flag*)
Control what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If *flag* is `False`, the cursor is left on the bottom line. If *flag* is `True`, the window is scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call *idlok()*.

window.**setscrreg**(*top*, *bottom*)
Set the scrolling region from line *top* to line *bottom*. All scrolling actions will take place in this region.

window.**standend**()
Turn off the standout attribute. On some terminals this has the side effect of turning off all attributes.

window.**standout**()
Turn on attribute *A_STANDOUT*.

window.**subpad**(*begin_y*, *begin_x*)
window.**subpad**(*nlines*, *ncols*, *begin_y*, *begin_x*)
Return a sub-window, whose upper-left corner is at `(begin_y, begin_x)`, and whose width/height is *ncols*/*nlines*.

window.**subwin**(*begin_y*, *begin_x*)
window.**subwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)
Return a sub-window, whose upper-left corner is at `(begin_y, begin_x)`, and whose width/height is *ncols*/*nlines*.

By default, the sub-window will extend from the specified position to the lower right corner of the window.

window.**syncdown**()
Touch each location in the window that has been touched in any of its ancestor windows. This routine is called by *refresh()*, so it should almost never be necessary to call it manually.

window.**syncok**(*flag*)
If *flag* is `True`, then *syncup()* is called automatically whenever there is a change in the window.

window.**syncup**()
Touch all locations in ancestors of the window that have been changed in the window.

window.**timeout**(*delay*)
Set blocking or non-blocking read behavior for the window. If *delay* is negative, blocking read is used (which will wait indefinitely for input). If *delay* is zero, then non-blocking read is used, and *getch()* will return `-1` if no input is waiting. If *delay* is positive, then *getch()* will block for *delay* milliseconds, and return `-1` if there is still no input at the end of that time.

window.**touchline**(*start*, *count*[, *changed*])
Pretend *count* lines have been changed, starting with line *start*. If *changed* is supplied, it specifies whether the affected lines are marked as having been changed (*changed*=`True`) or unchanged (*changed*=`False`).

window.**touchwin**()
Pretend the whole window has been changed, for purposes of drawing optimizations.

window.**untouchwin**()
Mark all lines in the window as unchanged since the last call to *refresh()*.

window.**vline**(*ch*, *n*)
window.**vline**(*y*, *x*, *ch*, *n*)
Display a vertical line starting at `(y, x)` with length *n* consisting of the character *ch*.

16.10.3 Constantes

The *curses* module defines the following data members:

curses.**ERR**
Some curses routines that return an integer, such as *getch()*, return *ERR* upon failure.

curses.**OK**
Some curses routines that return an integer, such as *napms()*, return *OK* upon success.

curses.**version**
A bytes object representing the current version of the module. Also available as `__version__`.

Some constants are available to specify character cell attributes. The exact constants available are system dependent.

Atributo	Significado
A_ALTCHARSET	Alternate character set mode
A_BLINK	Blink mode
A_BOLD	Bold mode
A_DIM	Dim mode
A_INVIS	Invisible or blank mode
A_ITALIC	Italic mode
A_NORMAL	Normal attribute
A_PROTECT	Protected mode
A_REVERSE	Reverse background and foreground colors
A_STANDOUT	Standout mode
A_UNDERLINE	Underline mode
A_HORIZONTAL	Horizontal highlight
A_LEFT	Left highlight
A_LOW	Low highlight
A_RIGHT	Right highlight
A_TOP	Top highlight
A_VERTICAL	Vertical highlight
A_CHARTEXT	Bit-mask to extract a character

Novo na versão 3.7: A_ITALIC was added.

Several constants are available to extract corresponding attributes returned by some methods.

Bit-mask	Significado
A_ATTRIBUTES	Bit-mask to extract attributes
A_CHARTEXT	Bit-mask to extract a character
A_COLOR	Bit-mask to extract color-pair field information

Keys are referred to by integer constants with names starting with KEY_. The exact keycaps available are system dependent.

Key constant	Chave
KEY_MIN	Minimum key value
KEY_BREAK	Break key (unreliable)
KEY_DOWN	Down-arrow
KEY_UP	Up-arrow
KEY_LEFT	Left-arrow
KEY_RIGHT	Right-arrow
KEY_HOME	Home key (upward+left arrow)
KEY_BACKSPACE	Backspace (unreliable)
KEY_F0	Function keys. Up to 64 function keys are supported.
KEY_Fn	Value of function key *n*
KEY_DL	Delete line
KEY_IL	Insert line
KEY_DC	Delete character
KEY_IC	Insert char or enter insert mode
KEY_EIC	Exit insert char mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen
KEY_EOL	Clear to end of line
KEY_SF	Scroll 1 line forward
KEY_SR	Scroll 1 line backward (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page
KEY_STAB	Set tab
KEY_CTAB	Clear tab
KEY_CATAB	Clear all tabs
KEY_ENTER	Enter or send (unreliable)
KEY_SRESET	Soft (partial) reset (unreliable)
KEY_RESET	Reset or hard reset (unreliable)
KEY_PRINT	Print
KEY_LL	Home down or bottom (lower left)
KEY_A1	Upper left of keypad
KEY_A3	Upper right of keypad
KEY_B2	Center of keypad
KEY_C1	Lower left of keypad
KEY_C3	Lower right of keypad
KEY_BTAB	Back tab
KEY_BEG	Beg (beginning)
KEY_CANCEL	Cancel
KEY_CLOSE	Close
KEY_COMMAND	Cmd (command)
KEY_COPY	Copy
KEY_CREATE	Create
KEY_END	End
KEY_EXIT	Exit
KEY_FIND	Find
KEY_HELP	Help
KEY_MARK	Mark
KEY_MESSAGE	Message
KEY_MOVE	Move
KEY_NEXT	Next

Continuação na próxima página

Tabela 1 – continuação da página anterior

Key constant	Chave
KEY_OPEN	Open
KEY_OPTIONS	Opções
KEY_PREVIOUS	Prev (previous)
KEY_REDO	Redo
KEY_REFERENCE	Ref (reference)
KEY_REFRESH	Refresh
KEY_REPLACE	Replace
KEY_RESTART	Restart
KEY_RESUME	Resume
KEY_SAVE	Salvar
KEY_SBEG	Shifted Beg (beginning)
KEY_SCANCEL	Shifted Cancel
KEY_SCOMMAND	Shifted Command
KEY_SCOPY	Shifted Copy
KEY_SCREATE	Shifted Create
KEY_SDC	Shifted Delete char
KEY_SDL	Shifted Delete line
KEY_SELECT	Select
KEY_SEND	Shifted End
KEY_SEOL	Shifted Clear line
KEY_SEXIT	Shifted Exit
KEY_SFIND	Shifted Find
KEY_SHELP	Shifted Help
KEY_SHOME	Shifted Home
KEY_SIC	Shifted Input
KEY_SLEFT	Shifted Left arrow
KEY_SMESSAGE	Shifted Message
KEY_SMOVE	Shifted Move
KEY_SNEXT	Shifted Next
KEY_SOPTIONS	Shifted Options
KEY_SPREVIOUS	Shifted Prev
KEY_SPRINT	Shifted Print
KEY_SREDO	Shifted Redo
KEY_SREPLACE	Shifted Replace
KEY_SRIGHT	Shifted Right arrow
KEY_SRSUME	Resumo alterado
KEY_SSAVE	Shifted Save
KEY_SSUSPEND	Shifted Suspend
KEY_SUNDO	Shifted Undo
KEY_SUSPEND	Suspend
KEY_UNDO	Desfazer
KEY_MOUSE	Mouse event has occurred
KEY_RESIZE	Terminal resize event
KEY_MAX	Maximum key value

On VT100s and their software emulations, such as X terminal emulators, there are normally at least four function keys (KEY_F1, KEY_F2, KEY_F3, KEY_F4) available, and the arrow keys mapped to KEY_UP, KEY_DOWN, KEY_LEFT and KEY_RIGHT in the obvious way. If your machine has a PC keyboard, it is safe to expect arrow keys and twelve function keys (older PC keyboards may have only ten function keys); also, the following keypad mappings are standard:

Keycap	Constante
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_PPAGE
Page Down	KEY_NPAGE

The following table lists characters from the alternate character set. These are inherited from the VT100 terminal, and will generally be available on software emulations such as X terminals. When there is no graphic available, curses falls back on a crude printable ASCII approximation.

Nota: These are available only after *initscr()* has been called.

ACS code	Significado
ACS_BBSS	alternate name for upper right corner
ACS_BLOCK	solid square block
ACS_BOARD	board of squares
ACS_BSBS	alternate name for horizontal line
ACS_BSSB	alternate name for upper left corner
ACS_BSSS	alternate name for top tee
ACS_BTEE	bottom tee
ACS_BULLET	bullet
ACS_CKBOARD	checker board (stipple)
ACS_DARROW	arrow pointing down
ACS_DEGREE	degree symbol
ACS_DIAMOND	diamond
ACS_GEQUAL	greater-than-or-equal-to
ACS_HLINE	horizontal line
ACS_LANTERN	lantern symbol
ACS_LARROW	left arrow
ACS_LEQUAL	less-than-or-equal-to
ACS_LLCORNER	lower left-hand corner
ACS_LRCORNER	lower right-hand corner
ACS_LTEE	left tee
ACS_NEQUAL	not-equal sign
ACS_PI	letter pi
ACS_PLMINUS	plus-or-minus sign
ACS_PLUS	big plus sign
ACS_RARROW	right arrow
ACS_RTEE	right tee
ACS_S1	scan line 1
ACS_S3	scan line 3
ACS_S7	scan line 7
ACS_S9	scan line 9
ACS_SBBS	alternate name for lower right corner
ACS_SBSB	alternate name for vertical line
ACS_SBSS	alternate name for right tee
ACS_SSBB	alternate name for lower left corner

Continuação na próxima página

Tabela 2 – continuação da página anterior

ACS code	Significado
ACS_SSBS	alternate name for bottom tee
ACS_SSSB	alternate name for left tee
ACS_SSSS	alternate name for crossover or big plus
ACS_STERLING	pound sterling
ACS_TTEE	top tee
ACS_UARROW	up arrow
ACS_ULCORNER	upper left corner
ACS_URCORNER	upper right corner
ACS_VLINE	vertical line

The following table lists the predefined colors:

Constante	Color
COLOR_BLACK	Black
COLOR_BLUE	Blue
COLOR_CYAN	Cyan (light greenish blue)
COLOR_GREEN	Green
COLOR_MAGENTA	Magenta (purplish red)
COLOR_RED	Red
COLOR_WHITE	White
COLOR_YELLOW	Yellow

16.11 curses.textpad — Text input widget for curses programs

The *curses.textpad* module provides a *Textbox* class that handles elementary text editing in a curses window, supporting a set of keybindings resembling those of Emacs (thus, also of Netscape Navigator, BBedit 6.x, FrameMaker, and many other programs). The module also provides a rectangle-drawing function useful for framing text boxes or for other purposes.

The module *curses.textpad* defines the following function:

curses.textpad.**rectangle**(*win*, *uly*, *ulx*, *lry*, *lrx*)

Draw a rectangle. The first argument must be a window object; the remaining arguments are coordinates relative to that window. The second and third arguments are the y and x coordinates of the upper left hand corner of the rectangle to be drawn; the fourth and fifth arguments are the y and x coordinates of the lower right hand corner. The rectangle will be drawn using VT100/IBM PC forms characters on terminals that make this possible (including xterm and most other software terminal emulators). Otherwise it will be drawn with ASCII dashes, vertical bars, and plus signs.

16.11.1 Textbox objects

You can instantiate a *Textbox* object as follows:

class curses.textpad.**Textbox**(*win*)

Return a textbox widget object. The *win* argument should be a curses *window* object in which the textbox is to be contained. The edit cursor of the textbox is initially located at the upper left hand corner of the containing window, with coordinates `(0, 0)`. The instance's *stripspaces* flag is initially on.

Textbox objects have the following methods:

edit([*validator*])
This is the entry point you will normally use. It accepts editing keystrokes until one of the termination keystrokes is entered. If *validator* is supplied, it must be a function. It will be called for each keystroke entered with the keystroke as a parameter; command dispatch is done on the result. This method returns the window contents as a string; whether blanks in the window are included is affected by the *stripspaces* attribute.

do_command(*ch*)
Process a single command keystroke. Here are the supported special keystrokes:

Keystroke	Action
Control-A	Go to left edge of window.
Control-B	Cursor left, wrapping to previous line if appropriate.
Control-D	Delete character under cursor.
Control-E	Go to right edge (stripspaces off) or end of line (stripspaces on).
Control-F	Cursor right, wrapping to next line when appropriate.
Control-G	Terminate, returning the window contents.
Control-H	Delete character backward.
Control-J	Terminate if the window is 1 line, otherwise insert newline.
Control-K	If line is blank, delete it, otherwise clear to end of line.
Control-L	Refresh screen.
Control-N	Cursor down; move down one line.
Control-O	Insert a blank line at cursor location.
Control-P	Cursor up; move up one line.

Move operations do nothing if the cursor is at an edge where the movement is not possible. The following synonyms are supported where possible:

Constante	Keystroke
KEY_LEFT	Control-B
KEY_RIGHT	Control-F
KEY_UP	Control-P
KEY_DOWN	Control-N
KEY_BACKSPACE	Control-h

All other keystrokes are treated as a command to insert the given character and move right (with line wrapping).

gather()
Return the window contents as a string; whether blanks in the window are included is affected by the *stripspaces* member.

stripspaces
This attribute is a flag which controls the interpretation of blanks in the window. When it is on, trailing blanks on each line are ignored; any cursor motion that would land the cursor on a trailing blank goes to the end of that line instead, and trailing blanks are stripped when the window contents are gathered.

16.12 curses.ascii — Utilities for ASCII characters

The *curses.ascii* module supplies name constants for ASCII characters and functions to test membership in various ASCII character classes. The constants supplied are names for control characters as follows:

Nome	Significado
NUL	
SOH	Start of heading, console interrupt
STX	Start of text
ETX	End of text
EOT	End of transmission
ENQ	Enquiry, goes with ACK flow control
ACK	Acknowledgement
BEL	Bell
BS	Backspace
TAB	Tab
HT	Alias for TAB: “Horizontal tab”
LF	Line feed
NL	Alias for LF: “New line”
VT	Vertical tab
FF	Form feed
CR	Carriage return
SO	Shift-out, begin alternate character set
SI	Shift-in, resume default character set
DLE	Data-link escape
DC1	XON, for flow control
DC2	Device control 2, block-mode flow control
DC3	XOFF, for flow control
DC4	Device control 4
NAK	Negative acknowledgement
SYN	Synchronous idle
ETB	End transmission block
CAN	Cancel
EM	End of medium
SUB	Substitute
ESC	Escape
FS	File separator
GS	Group separator
RS	Record separator, block-mode terminator
US	Separador de Unidade
SP	Espaço
DEL	Delete

Note that many of these have little practical significance in modern usage. The mnemonics derive from teleprinter conventions that predate digital computers.

The module supplies the following functions, patterned on those in the standard C library:

curses.ascii.**isalnum**(*c*)
Checks for an ASCII alphanumeric character; it is equivalent to `isalpha(c) or isdigit(c)`.

curses.ascii.**isalpha**(*c*)
Checks for an ASCII alphabetic character; it is equivalent to `isupper(c) or islower(c)`.

curses.ascii.**isascii**(*c*)
Checks for a character value that fits in the 7-bit ASCII set.

curses.ascii.**isblank**(*c*)
Checks for an ASCII whitespace character; space or horizontal tab.

curses.ascii.**iscntrl**(*c*)
Checks for an ASCII control character (in the range 0x00 to 0x1f or 0x7f).

curses.ascii.**isdigit**(*c*)
Checks for an ASCII decimal digit, `'0'` through `'9'`. This is equivalent to `c in string.digits`.

curses.ascii.**isgraph**(*c*)
Checks for ASCII any printable character except space.

curses.ascii.**islower**(*c*)
Checks for an ASCII lower-case character.

curses.ascii.**isprint**(*c*)
Checks for any ASCII printable character including space.

curses.ascii.**ispunct**(*c*)
Checks for any printable ASCII character which is not a space or an alphanumeric character.

curses.ascii.**isspace**(*c*)
Checks for ASCII white-space characters; space, line feed, carriage return, form feed, horizontal tab, vertical tab.

curses.ascii.**isupper**(*c*)
Checks for an ASCII uppercase letter.

curses.ascii.**isxdigit**(*c*)
Checks for an ASCII hexadecimal digit. This is equivalent to `c in string.hexdigits`.

curses.ascii.**isctrl**(*c*)
Checks for an ASCII control character (ordinal values 0 to 31).

curses.ascii.**ismeta**(*c*)
Checks for a non-ASCII character (ordinal values 0x80 and above).

These functions accept either integers or single-character strings; when the argument is a string, it is first converted using the built-in function *ord()*.

Note that all these functions check ordinal bit values derived from the character of the string you pass in; they do not actually know anything about the host machine's character encoding.

The following two functions take either a single-character string or integer byte value; they return a value of the same type.

curses.ascii.**ascii**(*c*)
Return the ASCII value corresponding to the low 7 bits of *c*.

curses.ascii.**ctrl**(*c*)
Return the control character corresponding to the given character (the character bit value is bitwise-anded with 0x1f).

curses.ascii.**alt**(*c*)
Return the 8-bit character corresponding to the given ASCII character (the character bit value is bitwise-ored with 0x80).

The following function takes either a single-character string or integer value; it returns a string.

curses.ascii.**unctrl**(*c*)
Return a string representation of the ASCII character *c*. If *c* is printable, this string is the character itself. If the character is a control character (0x00–0x1f) the string consists of a caret (`'^'`) followed by the corresponding uppercase letter. If the character is an ASCII delete (0x7f) the string is `'^?'`. If the character has its meta bit (0x80) set, the meta bit is stripped, the preceding rules applied, and `'!'` prepended to the result.

curses.ascii.**controlnames**
A 33-element string array that contains the ASCII mnemonics for the thirty-two ASCII control characters from 0 (NUL) to 0x1f (US), in order, plus the mnemonic `SP` for the space character.

16.13 `curses.panel` — A panel stack extension for curses

Panels are windows with the added feature of depth, so they can be stacked on top of each other, and only the visible portions of each window will be displayed. Panels can be added, moved up or down in the stack, and removed.

16.13.1 Funções

The module *curses.panel* defines the following functions:

curses.panel.**bottom_panel**()
Retorna o painel inferior da pilha de painéis.

curses.panel.**new_panel**(*win*)
Returns a panel object, associating it with the given window *win*. Be aware that you need to keep the returned panel object referenced explicitly. If you don't, the panel object is garbage collected and removed from the panel stack.

curses.panel.**top_panel**()
Returns the top panel in the panel stack.

curses.panel.**update_panels**()
Updates the virtual screen after changes in the panel stack. This does not call *curses.doupdate()*, so you'll have to do this yourself.

16.13.2 Objetos Panel

Panel objects, as returned by *new_panel()* above, are windows with a stacking order. There's always a window associated with a panel which determines the content, while the panel methods are responsible for the window's depth in the panel stack.

Objetos Panel possuem os seguintes métodos:

Panel.**above**()
Returns the panel above the current panel.

Panel.**below**()
Retorna o painel abaixo do painel atual.

Panel.**bottom**()
Push the panel to the bottom of the stack.

Panel.**hidden**()
Returns `True` if the panel is hidden (not visible), `False` otherwise.

`Panel.`**`hide`**`()`
Hide the panel. This does not delete the object, it just makes the window on screen invisible.

`Panel.`**`move`**`(`*y*, *x*`)`
Move the panel to the screen coordinates `(y, x)`.

`Panel.`**`replace`**`(`*win*`)`
Change the window associated with the panel to the window *win*.

`Panel.`**`set_userptr`**`(`*obj*`)`
Set the panel's user pointer to *obj*. This is used to associate an arbitrary piece of data with the panel, and can be any Python object.

`Panel.`**`show`**`()`
Display the panel (which might have been hidden).

`Panel.`**`top`**`()`
Push panel to the top of the stack.

`Panel.`**`userptr`**`()`
Retorna o ponteiro do usuário para o painel. Pode ser qualquer objeto Python.

`Panel.`**`window`**`()`
Returns the window object associated with the panel.

16.14 `platform` — Access to underlying platform's identifying data

Código Fonte: Lib/platform.py

Nota: Specific platforms listed alphabetically, with Linux included in the Unix section.

16.14.1 Cross Platform

`platform.`**`architecture`**`(`*executable=sys.executable*, *bits=''*, *linkage=''*`)`
Queries the given executable (defaults to the Python interpreter binary) for various architecture information.

Returns a tuple `(bits, linkage)` which contain information about the bit architecture and the linkage format used for the executable. Both values are returned as strings.

Values that cannot be determined are returned as given by the parameter presets. If bits is given as `''`, the `sizeof(pointer)` (or `sizeof(long)` on Python version < 1.5.2) is used as indicator for the supported pointer size.

The function relies on the system's `file` command to do the actual work. This is available on most if not all Unix platforms and some non-Unix platforms and then only if the executable points to the Python interpreter. Reasonable defaults are used when the above needs are not met.

Nota: On Mac OS X (and perhaps other platforms), executable files may be universal files containing multiple architectures.

To get at the "64-bitness" of the current interpreter, it is more reliable to query the *`sys.maxsize`* attribute:

```
is_64bits = sys.maxsize > 2**32
```

platform.**machine**()
Returns the machine type, e.g. 'i386'. An empty string is returned if the value cannot be determined.

platform.**node**()
Returns the computer's network name (may not be fully qualified!). An empty string is returned if the value cannot be determined.

platform.**platform**(*aliased=0*, *terse=0*)
Returns a single string identifying the underlying platform with as much useful information as possible.

The output is intended to be *human readable* rather than machine parseable. It may look different on different platforms and this is intended.

If *aliased* is true, the function will use aliases for various platforms that report system names which differ from their common names, for example SunOS will be reported as Solaris. The *system_alias()* function is used to implement this.

Setting *terse* to true causes the function to return only the absolute minimum information needed to identify the platform.

platform.**processor**()
Returns the (real) processor name, e.g. 'amdk6'.

An empty string is returned if the value cannot be determined. Note that many platforms do not provide this information or simply return the same value as for *machine()*. NetBSD does this.

platform.**python_build**()
Returns a tuple (buildno, builddate) stating the Python build number and date as strings.

platform.**python_compiler**()
Returns a string identifying the compiler used for compiling Python.

platform.**python_branch**()
Returns a string identifying the Python implementation SCM branch.

platform.**python_implementation**()
Returns a string identifying the Python implementation. Possible return values are: 'CPython', 'IronPython', 'Jython', 'PyPy'.

platform.**python_revision**()
Returns a string identifying the Python implementation SCM revision.

platform.**python_version**()
Returns the Python version as string 'major.minor.patchlevel'.

Note that unlike the Python sys.version, the returned value will always include the patchlevel (it defaults to 0).

platform.**python_version_tuple**()
Returns the Python version as tuple (major, minor, patchlevel) of strings.

Note that unlike the Python sys.version, the returned value will always include the patchlevel (it defaults to '0').

platform.**release**()
Returns the system's release, e.g. '2.2.0' or 'NT' An empty string is returned if the value cannot be determined.

platform.**system**()
Returns the system/OS name, such as 'Linux', 'Darwin', 'Java', 'Windows'. An empty string is returned if the value cannot be determined.

platform.**system_alias**(*system*, *release*, *version*)
Returns `(system, release, version)` aliased to common marketing names used for some systems. It also does some reordering of the information in some cases where it would otherwise cause confusion.

platform.**version**()
Returns the system's release version, e.g. `'#3 on degas'`. An empty string is returned if the value cannot be determined.

platform.**uname**()
Fairly portable uname interface. Returns a *namedtuple()* containing six attributes: *system*, *node*, *release*, *version*, *machine*, and *processor*.

Note that this adds a sixth attribute (*processor*) not present in the *os.uname()* result. Also, the attribute names are different for the first two attributes; *os.uname()* names them `sysname` and `nodename`.

Entries which cannot be determined are set to `''`.

Alterado na versão 3.3: Resultado alterado de uma tupla para uma namedtuple.

16.14.2 Java Platform

platform.**java_ver**(*release=''*, *vendor=''*, *vminfo=('', '', '')*, *osinfo=('', '', '')*)
Interface de versão para Jython.

Returns a tuple `(release, vendor, vminfo, osinfo)` with *vminfo* being a tuple `(vm_name, vm_release, vm_vendor)` and *osinfo* being a tuple `(os_name, os_version, os_arch)`. Values which cannot be determined are set to the defaults given as parameters (which all default to `''`).

16.14.3 Windows Platform

platform.**win32_ver**(*release=''*, *version=''*, *csd=''*, *ptype=''*)
Get additional version information from the Windows Registry and return a tuple `(release, version, csd, ptype)` referring to OS release, version number, CSD level (service pack) and OS type (multi/single processor).

As a hint: *ptype* is `'Uniprocessor Free'` on single processor NT machines and `'Multiprocessor Free'` on multi processor machines. The *'Free'* refers to the OS version being free of debugging code. It could also state *'Checked'* which means the OS version uses debugging code, i.e. code that checks arguments, ranges, etc.

Nota: This function works best with Mark Hammond's `win32all` package installed, but also on Python 2.3 and later (support for this was added in Python 2.6). It obviously only runs on Win32 compatible platforms.

Win95/98 specific

platform.**popen**(*cmd*, *mode='r'*, *bufsize=-1*)
Portable *popen()* interface. Find a working popen implementation preferring `win32pipe.popen()`. On Windows NT, `win32pipe.popen()` should work; on Windows 9x it hangs due to bugs in the MS C library.

Obsoleto desde a versão 3.3: This function is obsolete. Use the *subprocess* module. Check especially the *Replacing Older Functions with the subprocess Module* section.

16.14.4 Plataforma Mac OS

platform.**mac_ver**(*release=''*, *versioninfo=('', '', '')*, *machine=''*)
Get Mac OS version information and return it as tuple `(release, versioninfo, machine)` with *versioninfo* being a tuple `(version, dev_stage, non_release_version)`.

Entries which cannot be determined are set to `''`. All tuple entries are strings.

16.14.5 Plataformas Unix

platform.**dist**(*distname=''*, *version=''*, *id=''*, *supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...)*)
This is another name for *linux_distribution()*.

Deprecated since version 3.5, will be removed in version 3.8: See alternative like the distro package.

platform.**linux_distribution**(*distname=''*, *version=''*, *id=''*, *supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...)*, *full_distribution_name=1*)
Tries to determine the name of the Linux OS distribution name.

`supported_dists` may be given to define the set of Linux distributions to look for. It defaults to a list of currently supported Linux distributions identified by their release file name.

If `full_distribution_name` is true (default), the full distribution read from the OS is returned. Otherwise the short name taken from `supported_dists` is used.

Returns a tuple `(distname,version,id)` which defaults to the args given as parameters. `id` is the item in parentheses after the version number. It is usually the version codename.

Deprecated since version 3.5, will be removed in version 3.8: See alternative like the distro package.

platform.**libc_ver**(*executable=sys.executable*, *lib=''*, *version=''*, *chunksize=16384*)
Tries to determine the libc version against which the file executable (defaults to the Python interpreter) is linked. Returns a tuple of strings `(lib, version)` which default to the given parameters in case the lookup fails.

Note that this function has intimate knowledge of how different libc versions add symbols to the executable is probably only usable for executables compiled using **gcc**.

The file is read and scanned in chunks of *chunksize* bytes.

16.15 errno — Standard errno system symbols

This module makes available standard `errno` system symbols. The value of each symbol is the corresponding integer value. The names and descriptions are borrowed from `linux/include/errno.h`, which should be pretty all-inclusive.

errno.**errorcode**
Dictionary providing a mapping from the errno value to the string name in the underlying system. For instance, `errno.errorcode[errno.EPERM]` maps to `'EPERM'`.

To translate a numeric error code to an error message, use *os.strerror()*.

Of the following list, symbols that are not used on the current platform are not defined by the module. The specific list of defined symbols is available as `errno.errorcode.keys()`. Symbols available can include:

errno.**EPERM**
Operação não permitida

errno.**ENOENT**
No such file or directory

errno.**ESRCH**
No such process

errno.**EINTR**
Interrupted system call.

Ver também:

This error is mapped to the exception *InterruptedError*.

errno.**EIO**
I/O error

errno.**ENXIO**
No such device or address

errno.**E2BIG**
Arg list too long

errno.**ENOEXEC**
Exec format error

errno.**EBADF**
Bad file number

errno.**ECHILD**
No child processes

errno.**EAGAIN**
Try again

errno.**ENOMEM**
Out of memory

errno.**EACCES**
Permission denied

errno.**EFAULT**
Bad address

errno.**ENOTBLK**
Block device required

errno.**EBUSY**
Dispositivo ou recurso ocupado

errno.**EEXIST**
File exists

errno.**EXDEV**
Cross-device link

errno.**ENODEV**
No such device

errno.**ENOTDIR**
Not a directory

errno.**EISDIR**
Is a directory

errno.**EINVAL**
Invalid argument

errno.**ENFILE**
File table overflow

errno.**EMFILE**
Too many open files

errno.**ENOTTY**
Not a typewriter

errno.**ETXTBSY**
Text file busy

errno.**EFBIG**
File too large

errno.**ENOSPC**
No space left on device

errno.**ESPIPE**
Illegal seek

errno.**EROFS**
Read-only file system

errno.**EMLINK**
Too many links

errno.**EPIPE**
Broken pipe

errno.**EDOM**
Math argument out of domain of func

errno.**ERANGE**
Math result not representable

errno.**EDEADLK**
Resource deadlock would occur

errno.**ENAMETOOLONG**
File name too long

errno.**ENOLCK**
No record locks available

errno.**ENOSYS**
Function not implemented

errno.**ENOTEMPTY**
Directory not empty

errno.**ELOOP**
Too many symbolic links encountered

errno.**EWOULDBLOCK**
Operation would block

errno.**ENOMSG**
No message of desired type

errno.**EIDRM**
Identifier removed

errno.**ECHRNG**
Channel number out of range

errno.**EL2NSYNC**
Level 2 not synchronized

errno.**EL3HLT**
Level 3 halted

errno.**EL3RST**
Level 3 reset

errno.**ELNRNG**
Link number out of range

errno.**EUNATCH**
Protocol driver not attached

errno.**ENOCSI**
No CSI structure available

errno.**EL2HLT**
Level 2 halted

errno.**EBADE**
Invalid exchange

errno.**EBADR**
Invalid request descriptor

errno.**EXFULL**
Exchange full

errno.**ENOANO**
No anode

errno.**EBADRQC**
Invalid request code

errno.**EBADSLT**
Invalid slot

errno.**EDEADLOCK**
File locking deadlock error

errno.**EBFONT**
Bad font file format

errno.**ENOSTR**
Device not a stream

errno.**ENODATA**
No data available

errno.**ETIME**
Timer expired

errno.**ENOSR**
Out of streams resources

errno.**ENONET**
Machine is not on the network

errno.**ENOPKG**
Pacote não instalado

errno.**EREMOTE**
O objeto é remoto

errno.**ENOLINK**
Link has been severed

errno.**EADV**
Advertise error

errno.**ESRMNT**
Erro Srmount

errno.**ECOMM**
Communication error on send

errno.**EPROTO**
Erro de Protocolo

errno.**EMULTIHOP**
Multihop attempted

errno.**EDOTDOT**
RFS specific error

errno.**EBADMSG**
Not a data message

errno.**EOVERFLOW**
Value too large for defined data type

errno.**ENOTUNIQ**
Name not unique on network

errno.**EBADFD**
File descriptor in bad state

errno.**EREMCHG**
Remote address changed

errno.**ELIBACC**
Can not access a needed shared library

errno.**ELIBBAD**
Accessing a corrupted shared library

errno.**ELIBSCN**
.lib section in a.out corrupted

errno.**ELIBMAX**
Attempting to link in too many shared libraries

errno.**ELIBEXEC**
Cannot exec a shared library directly

errno.**EILSEQ**
Illegal byte sequence

errno.**ERESTART**
Interrupted system call should be restarted

errno.**ESTRPIPE**
Streams pipe error

errno.**EUSERS**
Too many users

errno.**ENOTSOCK**
Socket operation on non-socket

errno.**EDESTADDRREQ**
Destination address required

errno.**EMSGSIZE**
Message too long

errno.**EPROTOTYPE**
Protocol wrong type for socket

errno.**ENOPROTOOPT**
Protocol not available

errno.**EPROTONOSUPPORT**
Protocol not supported

errno.**ESOCKTNOSUPPORT**
Socket type not supported

errno.**EOPNOTSUPP**
Operation not supported on transport endpoint

errno.**EPFNOSUPPORT**
Protocol family not supported

errno.**EAFNOSUPPORT**
Address family not supported by protocol

errno.**EADDRINUSE**
Address already in use

errno.**EADDRNOTAVAIL**
Cannot assign requested address

errno.**ENETDOWN**
Network is down

errno.**ENETUNREACH**
Network is unreachable

errno.**ENETRESET**
Network dropped connection because of reset

errno.**ECONNABORTED**
Software caused connection abort

errno.**ECONNRESET**
Connection reset by peer

errno.**ENOBUFS**
No buffer space available

errno.**EISCONN**
Transport endpoint is already connected

errno.**ENOTCONN**
Transport endpoint is not connected

errno.**ESHUTDOWN**
Cannot send after transport endpoint shutdown

errno.**ETOOMANYREFS**
Too many references: cannot splice

errno.**ETIMEDOUT**
Connection timed out

errno.**ECONNREFUSED**
Connection refused

errno.**EHOSTDOWN**
Host is down

errno.**EHOSTUNREACH**
No route to host

errno.**EALREADY**
Operation already in progress

errno.**EINPROGRESS**
Operation now in progress

errno.**ESTALE**
Stale NFS file handle

errno.**EUCLEAN**
Structure needs cleaning

errno.**ENOTNAM**
Not a XENIX named type file

errno.**ENAVAIL**
No XENIX semaphores available

errno.**EISNAM**
É um arquivo de tipo nomeado

errno.**EREMOTEIO**
Erro de E/S remoto

errno.**EDQUOT**
Quota exceeded

16.16 `ctypes` — Uma biblioteca de funções externas para o Python

`ctypes` é uma biblioteca de funções externas para Python. Ela fornece tipos de dados compatíveis com C e permite funções de chamada em DLLs ou bibliotecas compartilhadas. Ela pode ser usada para agrupar essas bibliotecas em Python puro.

16.16.1 Tutorial ctypes

Nota: Os exemplos de código neste tutorial usam *doctest* para garantir que eles realmente funcionem. Como algumas amostras de código se comportam de maneira diferente no Linux, Windows ou Mac OS X, elas contêm diretrizes de doctest nos comentários.

Note: Some code samples reference the ctypes *c_int* type. On platforms where `sizeof(long) == sizeof(int)` it is an alias to *c_long*. So, you should not be confused if *c_long* is printed if you would expect *c_int* — they are actually the same type.

Loading dynamic link libraries

ctypes exports the *cdll*, and on Windows *windll* and *oledll* objects, for loading dynamic link libraries.

You load libraries by accessing them as attributes of these objects. *cdll* loads libraries which export functions using the standard `cdecl` calling convention, while *windll* libraries call functions using the `stdcall` calling convention. *oledll* also uses the `stdcall` calling convention, and assumes the functions return a Windows `HRESULT` error code. The error code is used to automatically raise an *OSError* exception when the function call fails.

Alterado na versão 3.3: Windows errors used to raise *WindowsError*, which is now an alias of *OSError*.

Here are some examples for Windows. Note that `msvcrt` is the MS standard C library containing most standard C functions, and uses the cdecl calling convention:

```
>>> from ctypes import *
>>> print(windll.kernel32)
<WinDLL 'kernel32', handle ... at ...>
>>> print(cdll.msvcrt)
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

Windows appends the usual `.dll` file suffix automatically.

Nota: Accessing the standard C library through `cdll.msvcrt` will use an outdated version of the library that may be incompatible with the one being used by Python. Where possible, use native Python functionality, or else import and use the `msvcrt` module.

On Linux, it is required to specify the filename *including* the extension to load a library, so attribute access can not be used to load libraries. Either the `LoadLibrary()` method of the dll loaders should be used, or you should load the library by creating an instance of CDLL by calling the constructor:

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

Accessing functions from loaded dlls

Funções são acessadas como atributos de objetos dll:

```
>>> from ctypes import *
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.GetModuleHandleA)
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.MyOwnFunction)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

Note that win32 system dlls like `kernel32` and `user32` often export ANSI as well as UNICODE versions of a function. The UNICODE version is exported with an `W` appended to the name, while the ANSI version is exported with an `A` appended to the name. The win32 `GetModuleHandle` function, which returns a *module handle* for a given module name, has the following C prototype, and a macro is used to expose one of them as `GetModuleHandle` depending on whether UNICODE is defined or not:

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

windll does not try to select one of them by magic, you must access the version you need by specifying `GetModuleHandleA` or `GetModuleHandleW` explicitly, and then call it with bytes or string objects respectively.

Sometimes, dlls export functions with names which aren't valid Python identifiers, like `"??2@YAPAXI@Z"`. In this case you have to use *`getattr()`* to retrieve the function:

```
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
>>>
```

On Windows, some dlls export functions not by name but by ordinal. These functions can be accessed by indexing the dll object with the ordinal number:

```
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

Calling functions

You can call these functions like any other Python callable. This example uses the `time()` function, which returns system time in seconds since the Unix epoch, and the `GetModuleHandleA()` function, which returns a win32 module handle.

This example calls both functions with a `NULL` pointer (`None` should be used as the `NULL` pointer):

```
>>> print(libc.time(None))
1150640792
>>> print(hex(windll.kernel32.GetModuleHandleA(None)))
0x1d000000
>>>
```

ValueError is raised when you call an `stdcall` function with the `cdecl` calling convention, or vice versa:

```
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>>

>>> windll.msvcrt.printf(b"spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

To find out the correct calling convention you have to look into the C header file or the documentation for the function you want to call.

On Windows, *ctypes* uses win32 structured exception handling to prevent crashes from general protection faults when functions are called with invalid argument values:

```
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: exception: access violation reading 0x00000020
>>>
```

There are, however, enough ways to crash Python with *ctypes*, so you should be careful anyway. The *faulthandler* module can be helpful in debugging crashes (e.g. from segmentation faults produced by erroneous C library calls).

`None`, integers, bytes objects and (unicode) strings are the only native Python objects that can directly be used as parameters in these function calls. `None` is passed as a C `NULL` pointer, bytes objects and strings are passed as pointer to the memory block that contains their data (`char *` or `wchar_t *`). Python integers are passed as the platforms default C `int` type, their value is masked to fit into the C type.

Before we move on calling functions with other parameter types, we have to learn more about *ctypes* data types.

Fundamental data types

ctypes defines a number of primitive C compatible data types:

ctypes type	C type	Python type
c_bool	_Bool	bool (1)
c_char	char	1-character bytes object
c_wchar	wchar_t	1-character string
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned short	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 or long long	int
c_ulonglong	unsigned __int64 or unsigned long long	int
c_size_t	size_t	int
c_ssize_t	ssize_t or Py_ssize_t	int
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char * (NUL terminated)	bytes object ou None
c_wchar_p	wchar_t * (NUL terminated)	String ou None
c_void_p	void *	int ou None

(1) The constructor accepts any object with a truth value.

All these types can be created by calling them with an optional initializer of the correct type and value:

```
>>> c_int()
c_long(0)
>>> c_wchar_p("Hello, World")
c_wchar_p(140018365411392)
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

Since these types are mutable, their value can also be changed afterwards:

```
>>> i = c_int(42)
>>> print(i)
c_long(42)
>>> print(i.value)
42
>>> i.value = -99
>>> print(i.value)
-99
>>>
```

Assigning a new value to instances of the pointer types *c_char_p*, *c_wchar_p*, and *c_void_p* changes the *memory location* they point to, *not the contents* of the memory block (of course not, because Python bytes objects are immutable):

```
>>> s = "Hello, World"
>>> c_s = c_wchar_p(s)
>>> print(c_s)
c_wchar_p(139966785747344)
>>> print(c_s.value)
Hello World
>>> c_s.value = "Hi, there"
>>> print(c_s)              # the memory location has changed
c_wchar_p(139966783348904)
>>> print(c_s.value)
Hi, there
>>> print(s)                # first object is unchanged
Hello, World
>>>
```

You should be careful, however, not to pass them to functions expecting pointers to mutable memory. If you need mutable memory blocks, ctypes has a *create_string_buffer()* function which creates these in various ways. The current memory block contents can be accessed (or changed) with the `raw` property; if you want to access it as NUL terminated string, use the `value` property:

```
>>> from ctypes import *
>>> p = create_string_buffer(3)            # create a 3 byte buffer, initialized to␣
↪NUL bytes
>>> print(sizeof(p), repr(p.raw))
3 b'\x00\x00\x00'
>>> p = create_string_buffer(b"Hello")     # create a buffer containing a NUL␣
↪terminated string
>>> print(sizeof(p), repr(p.raw))
6 b'Hello\x00'
>>> print(repr(p.value))
b'Hello'
>>> p = create_string_buffer(b"Hello", 10) # create a 10 byte buffer
>>> print(sizeof(p), repr(p.raw))
10 b'Hello\x00\x00\x00\x00\x00'
>>> p.value = b"Hi"
>>> print(sizeof(p), repr(p.raw))
10 b'Hi\x00lo\x00\x00\x00\x00\x00'
>>>
```

The *create_string_buffer()* function replaces the `c_buffer()` function (which is still available as an alias), as well as the `c_string()` function from earlier ctypes releases. To create a mutable memory block containing unicode characters of the C type `wchar_t` use the *create_unicode_buffer()* function.

Invocação de Funções, continuação

Note that printf prints to the real standard output channel, *not* to *sys.stdout*, so these examples will only work at the console prompt, not from within *IDLE* or *PythonWin*:

```
>>> printf = libc.printf
>>> printf(b"Hello, %s\n", b"World!")
Hello, World!
14
>>> printf(b"Hello, %S\n", "World!")
Hello, World!
14
>>> printf(b"%d bottles of beer\n", 42)
```

(continua na próxima página)

(continuação da página anterior)

```
42 bottles of beer
19
>>> printf(b"%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: Don't know how to convert parameter 2
>>>
```

As has been mentioned before, all Python types except integers, strings, and bytes objects have to be wrapped in their corresponding *ctypes* type, so that they can be converted to the required C data type:

```
>>> printf(b"An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>>
```

Calling functions with your own custom data types

You can also customize *ctypes* argument conversion to allow instances of your own classes be used as function arguments. *ctypes* looks for an `_as_parameter_` attribute and uses this as the function argument. Of course, it must be one of integer, string, or bytes:

```
>>> class Bottles:
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf(b"%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

If you don't want to store the instance's data in the `_as_parameter_` instance variable, you could define a *property* which makes the attribute available on request.

Specifying the required argument types (function prototypes)

It is possible to specify the required argument types of functions exported from DLLs by setting the `argtypes` attribute.

`argtypes` must be a sequence of C data types (the `printf` function is probably not a good example here, because it takes a variable number and different types of parameters depending on the format string, on the other hand this is quite handy to experiment with this feature):

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf(b"String '%s', Int %d, Double %f\n", b"Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

Specifying a format protects against incompatible argument types (just as a prototype for a C function), and tries to convert the arguments to valid types:

```
>>> printf(b"%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: wrong type
>>> printf(b"%s %d %f\n", b"X", 2, 3)
X 2 3.000000
13
>>>
```

If you have defined your own classes which you pass to function calls, you have to implement a `from_param()` class method for them to be able to use them in the `argtypes` sequence. The `from_param()` class method receives the Python object passed to the function call, it should do a typecheck or whatever is needed to make sure this object is acceptable, and then return the object itself, its `_as_parameter_` attribute, or whatever you want to pass as the C function argument in this case. Again, the result should be an integer, string, bytes, a *ctypes* instance, or an object with an `_as_parameter_` attribute.

Tipos de Retorno

By default functions are assumed to return the C `int` type. Other return types can be specified by setting the `restype` attribute of the function object.

Here is a more advanced example, it uses the `strchr` function, which expects a string pointer and a char, and returns a pointer to a string:

```
>>> strchr = libc.strchr
>>> strchr(b"abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p     # c_char_p is a pointer to a string
>>> strchr(b"abcdef", ord("d"))
b'def'
>>> print(strchr(b"abcdef", ord("x")))
None
>>>
```

If you want to avoid the `ord("x")` calls above, you can set the `argtypes` attribute, and the second argument will be converted from a single character Python bytes object into a C char:

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
'def'
>>> strchr(b"abcdef", b"def")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: one character string expected
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
'def'
>>>
```

You can also use a callable Python object (a function or a class for example) as the `restype` attribute, if the foreign function returns an integer. The callable will be called with the *integer* the C function returns, and the result of this call will be used as the result of your function call. This is useful to check for error return values and automatically raise an exception:

```
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>>
>>> GetModuleHandle.restype = ValidHandle
>>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in ValidHandle
OSError: [Errno 126] The specified module could not be found.
>>>
```

`WinError` is a function which will call Windows `FormatMessage()` api to get the string representation of an error code, and *returns* an exception. `WinError` takes an optional error code parameter, if no one is used, it calls *`GetLastError()`* to retrieve it.

Please note that a much more powerful error checking mechanism is available through the `errcheck` attribute; see the reference manual for details.

Passing pointers (or: passing parameters by reference)

Sometimes a C api function expects a *pointer* to a data type as parameter, probably to write into the corresponding location, or if the data is too large to be passed by value. This is also known as *passing parameters by reference*.

`ctypes` exports the *`byref()`* function which is used to pass parameters by reference. The same effect can be achieved with the *`pointer()`* function, although *`pointer()`* does a lot more work since it constructs a real pointer object, so it is faster to use *`byref()`* if you don't need the pointer object in Python itself:

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(b'\000' * 32)
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
>>> libc.sscanf(b"1 3.14 Hello", b"%d %f %s",
...             byref(i), byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
1 3.1400001049 b'Hello'
>>>
```

Structures and unions

Structures and unions must derive from the *`Structure`* and *`Union`* base classes which are defined in the *`ctypes`* module. Each subclass must define a `_fields_` attribute. `_fields_` must be a list of *2-tuples*, containing a *field name* and a *field type*.

The field type must be a *`ctypes`* type like *`c_int`*, or any other derived *`ctypes`* type: structure, union, array, pointer.

Here is a simple example of a POINT structure, which contains two integers named *x* and *y*, and also shows how to initialize a structure in the constructor:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print(point.x, point.y)
10 20
>>> point = POINT(y=5)
>>> print(point.x, point.y)
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: too many initializers
>>>
```

You can, however, build much more complicated structures. A structure can itself contain other structures by using a structure as a field type.

Here is a RECT structure which contains two POINTs named *upperleft* and *lowerright*:

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
>>> rc = RECT(point)
>>> print(rc.upperleft.x, rc.upperleft.y)
0 5
>>> print(rc.lowerright.x, rc.lowerright.y)
0 0
>>>
```

Nested structures can also be initialized in the constructor in several ways:

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

Field *descriptor*s can be retrieved from the *class*, they are useful for debugging because they can provide useful information:

```
>>> print(POINT.x)
<Field type=c_long, ofs=0, size=4>
>>> print(POINT.y)
<Field type=c_long, ofs=4, size=4>
>>>
```

Aviso: `ctypes` does not support passing unions or structures with bit-fields to functions by value. While this may work on 32-bit x86, it's not guaranteed by the library to work in the general case. Unions and structures with bit-fields should always be passed to functions by pointer.

Structure/union alignment and byte order

By default, Structure and Union fields are aligned in the same way the C compiler does it. It is possible to override this behavior by specifying a _pack_ class attribute in the subclass definition. This must be set to a positive integer and specifies the maximum alignment for the fields. This is what `#pragma pack(n)` also does in MSVC.

ctypes uses the native byte order for Structures and Unions. To build structures with non-native byte order, you can use one of the *BigEndianStructure*, *LittleEndianStructure*, `BigEndianUnion`, and `LittleEndianUnion` base classes. These classes cannot contain pointer fields.

Bit fields in structures and unions

It is possible to create structures and unions containing bit fields. Bit fields are only possible for integer fields, the bit width is specified as the third item in the _fields_ tuples:

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print(Int.first_16)
<Field type=c_long, ofs=0:0, bits=16>
>>> print(Int.second_16)
<Field type=c_long, ofs=0:16, bits=16>
>>>
```

Arrays

Arrays are sequences, containing a fixed number of instances of the same type.

The recommended way to create array types is by multiplying a data type with a positive integer:

```
TenPointsArrayType = POINT * 10
```

Here is an example of a somewhat artificial data type, a structure containing 4 POINTs among other stuff:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
...     _fields_ = [("a", c_int),
...                 ("b", c_float),
...                 ("point_array", POINT * 4)]
>>>
>>> print(len(MyStruct().point_array))
4
>>>
```

Instances are created in the usual way, by calling the class:

```
arr = TenPointsArrayType()
for pt in arr:
    print(pt.x, pt.y)
```

The above code print a series of `0 0` lines, because the array contents is initialized to zeros.

Initializers of the correct type can also be specified:

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(ii)
<c_long_Array_10 object at 0x...>
>>> for i in ii: print(i, end=" ")
...
1 2 3 4 5 6 7 8 9 10
>>>
```

Ponteiros

Pointer instances are created by calling the *pointer()* function on a *ctypes* type:

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

Pointer instances have a *contents* attribute which returns the object to which the pointer points, the `i` object above:

```
>>> pi.contents
c_long(42)
>>>
```

Note that *ctypes* does not have OOR (original object return), it constructs a new, equivalent object each time you retrieve an attribute:

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

Assigning another *c_int* instance to the pointer's contents attribute would cause the pointer to point to the memory location where this is stored:

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

Pointer instances can also be indexed with integers:

```
>>> pi[0]
99
>>>
```

Assigning to an integer index changes the pointed to value:

```
>>> print(i)
c_long(99)
>>> pi[0] = 22
```

(continua na próxima página)

(continuação da página anterior)

```
>>> print(i)
c_long(22)
>>>
```

It is also possible to use indexes different from 0, but you must know what you're doing, just as in C: You can access or change arbitrary memory locations. Generally you only use this feature if you receive a pointer from a C function, and you *know* that the pointer actually points to an array instead of a single item.

Behind the scenes, the *pointer()* function does more than simply create pointer instances, it has to create pointer *types* first. This is done with the *POINTER()* function, which accepts any *ctypes* type, and returns a new type:

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
>>>
```

Calling the pointer type without an argument creates a `NULL` pointer. `NULL` pointers have a `False` boolean value:

```
>>> null_ptr = POINTER(c_int)()
>>> print(bool(null_ptr))
False
>>>
```

ctypes checks for `NULL` when dereferencing pointers (but dereferencing invalid non-`NULL` pointers would crash Python):

```
>>> null_ptr[0]
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>
```

Conversão de Tipos

Usually, ctypes does strict type checking. This means, if you have `POINTER(c_int)` in the `argtypes` list of a function or as the type of a member field in a structure definition, only instances of exactly the same type are accepted. There are some exceptions to this rule, where ctypes accepts other objects. For example, you can pass compatible array instances instead of pointer types. So, for `POINTER(c_int)`, ctypes accepts an array of c_int:

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))]
...
>>> bar = Bar()
```

(continua na próxima página)

(continuação da página anterior)

```
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print(bar.values[i])
...
1
2
3
>>>
```

In addition, if a function argument is explicitly declared to be a pointer type (such as `POINTER(c_int)`) in `argtypes`, an object of the pointed type (`c_int` in this case) can be passed to the function. ctypes will apply the required *byref()* conversion in this case automatically.

To set a POINTER type field to `NULL`, you can assign `None`:

```
>>> bar.values = None
>>>
```

Sometimes you have instances of incompatible types. In C, you can cast one type into another type. *ctypes* provides a *cast()* function which can be used in the same way. The `Bar` structure defined above accepts `POINTER(c_int)` pointers or *c_int* arrays for its `values` field, but not instances of other types:

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: incompatible types, c_byte_Array_4 instance instead of LP_c_long instance
>>>
```

For these cases, the *cast()* function is handy.

The *cast()* function can be used to cast a ctypes instance into a pointer to a different ctypes data type. *cast()* takes two parameters, a ctypes object that is or can be converted to a pointer of some kind, and a ctypes pointer type. It returns an instance of the second argument, which references the same memory block as the first argument:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

So, *cast()* can be used to assign to the `values` field of `Bar` the structure:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print(bar.values[0])
0
>>>
```

Tipos Incompletos

Incomplete Types are structures, unions or arrays whose members are not yet specified. In C, they are specified by forward declarations, which are defined later:

```
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

The straightforward translation into ctypes code would be this, but it does not work:

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

because the new `class cell` is not available in the class statement itself. In *ctypes*, we can define the `cell` class and set the `_fields_` attribute later, after the class statement:

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

Let's try it. We create two instances of `cell`, and let them point to each other, and finally follow the pointer chain a few times:

```
>>> c1 = cell()
>>> c1.name = "foo"
>>> c2 = cell()
>>> c2.name = "bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print(p.name, end=" ")
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

Funções Callbacks

ctypes allows creating C callable function pointers from Python callables. These are sometimes called *callback functions*.

First, you must create a class for the callback function. The class knows the calling convention, the return type, and the number and types of arguments this function will receive.

The *CFUNCTYPE()* factory function creates types for callback functions using the `cdecl` calling convention. On Windows, the *WINFUNCTYPE()* factory function creates types for callback functions using the `stdcall` calling convention.

Both of these factory functions are called with the result type as first argument, and the callback functions expected argument types as the remaining arguments.

I will present an example here which uses the standard C library's `qsort()` function, that is used to sort items with the help of a callback function. `qsort()` will be used to sort an array of integers:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

`qsort()` must be called with a pointer to the data to sort, the number of items in the data array, the size of one item, and a pointer to the comparison function, the callback. The callback will then be called with two pointers to items, and it must return a negative integer if the first item is smaller than the second, a zero if they are equal, and a positive integer otherwise.

So our callback function receives pointers to integers, and must return an integer. First we create the `type` for the callback function:

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

To get started, here is a simple callback that shows the values it gets passed:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

O resultado:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

Now we can actually compare the two items and return a useful result:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
```

(continua na próxima página)

(continuação da página anterior)

```
...     return a[0] - b[0]
...
>>>
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

As we can easily check, our array is sorted now:

```
>>> for i in ia: print(i, end=" ")
...
1 5 7 33 99
>>>
```

The function factories can be used as decorator factories, so we may as well write:

```
>>> @CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
... def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>> qsort(ia, len(ia), sizeof(c_int), py_cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

Nota: Make sure you keep references to *CFUNCTYPE()* objects as long as they are used from C code. *ctypes* doesn't, and if you don't, they may be garbage collected, crashing your program when a callback is made.

Also, note that if the callback function is called in a thread created outside of Python's control (e.g. by the foreign code that calls the callback), ctypes creates a new dummy Python thread on every invocation. This behavior is correct for most purposes, but it means that values stored with *threading.local* will *not* survive across different callbacks, even when those calls are made from the same C thread.

Accessing values exported from dlls

Some shared libraries not only export functions, they also export variables. An example in the Python library itself is the `Py_OptimizeFlag`, an integer set to 0, 1, or 2, depending on the `-O` or `-OO` flag given on startup.

ctypes can access values like this with the `in_dll()` class methods of the type. *pythonapi* is a predefined symbol giving access to the Python C api:

```
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print(opt_flag)
c_long(0)
>>>
```

If the interpreter would have been started with `-O`, the sample would have printed `c_long(1)`, or `c_long(2)` if `-OO` would have been specified.

An extended example which also demonstrates the use of pointers accesses the `PyImport_FrozenModules` pointer exported by Python.

Quoting the docs for that value:

> Este ponteiro é inicializado para apontar para um vetor de registros de `struct_frozen`, terminado por um cujos membros são todos `NULL` ou zero. Quando um módulo congelado é importado, ele é pesquisado nesta tabela. O código de terceiros pode fazer truques com isso para fornecer uma coleção criada dinamicamente de módulos congelados.

So manipulating this pointer could even prove useful. To restrict the example size, we show only how this table can be read with *ctypes*:

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int)]
...
>>>
```

We have defined the `struct _frozen` data type, so we can get the pointer to the table:

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_FrozenModules")
>>>
```

Since `table` is a `pointer` to the array of `struct_frozen` records, we can iterate over it, but we just have to make sure that our loop terminates, because pointers have no size. Sooner or later it would probably crash with an access violation or whatever, so it's better to break out of the loop when we hit the `NULL` entry:

```
>>> for item in table:
...     if item.name is None:
...         break
...     print(item.name.decode("ascii"), item.size)
...
_frozen_importlib 31764
_frozen_importlib_external 41499
__hello__ 161
__phello__ -161
__phello__.spam 161
>>>
```

The fact that standard Python has a frozen module and a frozen package (indicated by the negative `size` member) is not well known, it is only used for testing. Try it out with `import __hello__` for example.

Surprises

There are some edges in *ctypes* where you might expect something other than what actually happens.

Consider the following example:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
3 4 3 4
>>>
```

Hm. We certainly expected the last statement to print `3 4 1 2`. What happened? Here are the steps of the `rc.a, rc.b = rc.b, rc.a` line above:

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

Note that `temp0` and `temp1` are objects still using the internal buffer of the `rc` object above. So executing `rc.a = temp0` copies the buffer contents of `temp0` into `rc` 's buffer. This, in turn, changes the contents of `temp1`. So, the last assignment `rc.b = temp1`, doesn't have the expected effect.

Keep in mind that retrieving sub-objects from Structure, Unions, and Arrays doesn't *copy* the sub-object, instead it retrieves a wrapper object accessing the root-object's underlying buffer.

Another example that may behave differently from what one would expect is this:

```
>>> s = c_char_p()
>>> s.value = b"abc def ghi"
>>> s.value
b'abc def ghi'
>>> s.value is s.value
False
>>>
```

Nota: Objects instantiated from *c_char_p* can only have their value set to bytes or integers.

Why is it printing `False`? ctypes instances are objects containing a memory block plus some *descriptor*s accessing the contents of the memory. Storing a Python object in the memory block does not store the object itself, instead the `contents` of the object is stored. Accessing the contents again constructs a new Python object each time!

Variable-sized data types

ctypes provides some support for variable-sized arrays and structures.

The *resize()* function can be used to resize the memory buffer of an existing ctypes object. The function takes the object as first argument, and the requested size in bytes as the second argument. The memory block cannot be made smaller than the natural memory block specified by the objects type, a *ValueError* is raised if this is tried:

```
>>> short_array = (c_short * 4)()
>>> print(sizeof(short_array))
8
>>> resize(short_array, 4)
Traceback (most recent call last):
    ...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

This is nice and fine, but how would one access the additional elements contained in this array? Since the type still only knows about 4 elements, we get errors accessing other elements:

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
    ...
IndexError: invalid index
>>>
```

Another way to use variable-sized data types with *ctypes* is to use the dynamic nature of Python, and (re-)define the data type after the required size is already known, on a case by case basis.

16.16.2 Referência ctypes

Finding shared libraries

When programming in a compiled language, shared libraries are accessed when compiling/linking a program, and when the program is run.

The purpose of the `find_library()` function is to locate a library in a way similar to what the compiler or runtime loader does (on platforms with several versions of a shared library the most recent should be loaded), while the ctypes library loaders act like when a program is run, and call the runtime loader directly.

The `ctypes.util` module provides a function which can help to determine the library to load.

`ctypes.util.`**`find_library`**(*name*)
: Try to find a library and return a pathname. *name* is the library name without any prefix like *lib*, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option `-l`). If no library can be found, returns `None`.

The exact functionality is system dependent.

On Linux, `find_library()` tries to run external programs (`/sbin/ldconfig`, `gcc`, `objdump` and `ld`) to find the library file. It returns the filename of the library file.

Alterado na versão 3.6: On Linux, the value of the environment variable `LD_LIBRARY_PATH` is used when searching for libraries, if a library cannot be found by any other means.

Here are some examples:

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
>>>
```

On OS X, `find_library()` tries several predefined naming schemes and paths to locate the library, and returns a full pathname if successful:

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

On Windows, `find_library()` searches along the system search path, and returns the full pathname, but since there is no predefined naming scheme a call like `find_library("c")` will fail and return `None`.

If wrapping a shared library with *ctypes*, it *may* be better to determine the shared library name at development time, and hardcode that into the wrapper module instead of using `find_library()` to locate the library at runtime.

Loading shared libraries

There are several ways to load shared libraries into the Python process. One way is to instantiate one of the following classes:

class ctypes.**CDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*)

Instances of this class represent loaded shared libraries. Functions in these libraries use the standard C calling convention, and are assumed to return `int`.

class ctypes.**OleDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*)

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return the windows specific *HRESULT* code. *HRESULT* values contain information specifying whether the function call failed or succeeded, together with additional error code. If the return value signals a failure, an *OSError* is automatically raised.

Alterado na versão 3.3: *WindowsError* used to be raised.

class ctypes.**WinDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*)

Windows only: Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return `int` by default.

On Windows CE only the standard calling convention is used, for convenience the *WinDLL* and *OleDLL* use the standard calling convention on this platform.

The Python *global interpreter lock* is released before calling any function exported by these libraries, and reacquired afterwards.

class ctypes.**PyDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*)

Instances of this class behave like *CDLL* instances, except that the Python GIL is *not* released during the function call, and after the function execution the Python error flag is checked. If the error flag is set, a Python exception is raised.

Thus, this is only useful to call Python C api functions directly.

All these classes can be instantiated by calling them with at least one argument, the pathname of the shared library. If you have an existing handle to an already loaded shared library, it can be passed as the `handle` named parameter, otherwise the underlying platforms `dlopen` or `LoadLibrary` function is used to load the library into the process, and to get a handle to it.

The *mode* parameter can be used to specify how the library is loaded. For details, consult the *dlopen(3)* manpage. On Windows, *mode* is ignored. On posix systems, RTLD_NOW is always added, and is not configurable.

The *use_errno* parameter, when set to true, enables a ctypes mechanism that allows accessing the system *errno* error number in a safe way. *ctypes* maintains a thread-local copy of the systems *errno* variable; if you call foreign functions created with `use_errno=True` then the *errno* value before the function call is swapped with the ctypes private copy, the same happens immediately after the function call.

The function *ctypes.get_errno()* returns the value of the ctypes private copy, and the function *ctypes.set_errno()* changes the ctypes private copy to a new value and returns the former value.

The *use_last_error* parameter, when set to true, enables the same mechanism for the Windows error code which is managed by the *GetLastError()* and `SetLastError()` Windows API functions; *ctypes.get_last_error()* and *ctypes.set_last_error()* are used to request and change the ctypes private copy of the windows error code.

ctypes.**RTLD_GLOBAL**

Flag to use as *mode* parameter. On platforms where this flag is not available, it is defined as the integer zero.

ctypes.**RTLD_LOCAL**

Flag to use as *mode* parameter. On platforms where this is not available, it is the same as *RTLD_GLOBAL*.

ctypes.**DEFAULT_MODE**

The default mode which is used to load shared libraries. On OSX 10.3, this is *RTLD_GLOBAL*, otherwise it is the same as *RTLD_LOCAL*.

Instances of these classes have no public methods. Functions exported by the shared library can be accessed as attributes or by index. Please note that accessing the function through an attribute caches the result and therefore accessing it repeatedly returns the same object each time. On the other hand, accessing it through an index returns a new object each time:

```
>>> from ctypes import CDLL
>>> libc = CDLL("libc.so.6")  # On Linux
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

The following public attributes are available, their name starts with an underscore to not clash with exported function names:

PyDLL.**_handle**

The system handle used to access the library.

PyDLL.**_name**
The name of the library passed in the constructor.

Shared libraries can also be loaded by using one of the prefabricated objects, which are instances of the *LibraryLoader* class, either by calling the LoadLibrary() method, or by retrieving the library as attribute of the loader instance.

class ctypes.**LibraryLoader**(*dlltype*)
Class which loads shared libraries. *dlltype* should be one of the *CDLL*, *PyDLL*, *WinDLL*, or *OleDLL* types.

__getattr__() has special behavior: It allows loading a shared library by accessing it as attribute of a library loader instance. The result is cached, so repeated attribute accesses return the same library each time.

LoadLibrary(*name*)
Load a shared library into the process and return it. This method always returns a new instance of the library.

These prefabricated library loaders are available:

ctypes.**cdll**
Creates *CDLL* instances.

ctypes.**windll**
Windows only: Creates *WinDLL* instances.

ctypes.**oledll**
Windows only: Creates *OleDLL* instances.

ctypes.**pydll**
Creates *PyDLL* instances.

For accessing the C Python api directly, a ready-to-use Python shared library object is available:

ctypes.**pythonapi**
An instance of *PyDLL* that exposes Python C API functions as attributes. Note that all these functions are assumed to return C int, which is of course not always the truth, so you have to assign the correct restype attribute to use these functions.

Foreign functions

As explained in the previous section, foreign functions can be accessed as attributes of loaded shared libraries. The function objects created in this way by default accept any number of arguments, accept any ctypes data instances as arguments, and return the default result type specified by the library loader. They are instances of a private class:

class ctypes.**_FuncPtr**
Base class for C callable foreign functions.

Instances of foreign functions are also C compatible data types; they represent C function pointers.

This behavior can be customized by assigning to special attributes of the foreign function object.

restype
Assign a ctypes type to specify the result type of the foreign function. Use None for void, a function not returning anything.

It is possible to assign a callable Python object that is not a ctypes type, in this case the function is assumed to return a C int, and the callable will be called with this integer, allowing further processing or error checking. Using this is deprecated, for more flexible post processing or error checking use a ctypes data type as *restype* and assign a callable to the *errcheck* attribute.

argtypes
Assign a tuple of ctypes types to specify the argument types that the function accepts. Functions using the

`stdcall` calling convention can only be called with the same number of arguments as the length of this tuple; functions using the C calling convention accept additional, unspecified arguments as well.

When a foreign function is called, each actual argument is passed to the `from_param()` class method of the items in the *argtypes* tuple, this method allows adapting the actual argument to an object that the foreign function accepts. For example, a *c_char_p* item in the *argtypes* tuple will convert a string passed as argument into a bytes object using ctypes conversion rules.

New: It is now possible to put items in argtypes which are not ctypes types, but each item must have a `from_param()` method which returns a value usable as argument (integer, string, ctypes instance). This allows defining adapters that can adapt custom objects as function parameters.

errcheck
Assign a Python function or another callable to this attribute. The callable will be called with three or more arguments:

callable(*result*, *func*, *arguments*)
result is what the foreign function returns, as specified by the `restype` attribute.

func is the foreign function object itself, this allows reusing the same callable object to check or post process the results of several functions.

arguments is a tuple containing the parameters originally passed to the function call, this allows specializing the behavior on the arguments used.

The object that this function returns will be returned from the foreign function call, but it can also check the result value and raise an exception if the foreign function call failed.

exception ctypes.**ArgumentError**
This exception is raised when a foreign function call cannot convert one of the passed arguments.

Function prototypes

Foreign functions can also be created by instantiating function prototypes. Function prototypes are similar to function prototypes in C; they describe a function (return type, argument types, calling convention) without defining an implementation. The factory functions must be called with the desired result type and the argument types of the function, and can be used as decorator factories, and as such, be applied to functions through the `@wrapper` syntax. See *Funções Callbacks* for examples.

ctypes.**CFUNCTYPE**(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)
The returned function prototype creates functions that use the standard C calling convention. The function will release the GIL during the call. If *use_errno* is set to true, the ctypes private copy of the system *errno* variable is exchanged with the real *errno* value before and after the call; *use_last_error* does the same for the Windows error code.

ctypes.**WINFUNCTYPE**(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)
Windows only: The returned function prototype creates functions that use the `stdcall` calling convention, except on Windows CE where *WINFUNCTYPE()* is the same as *CFUNCTYPE()*. The function will release the GIL during the call. *use_errno* and *use_last_error* have the same meaning as above.

ctypes.**PYFUNCTYPE**(*restype*, **argtypes*)
The returned function prototype creates functions that use the Python calling convention. The function will *not* release the GIL during the call.

Function prototypes created by these factory functions can be instantiated in different ways, depending on the type and number of the parameters in the call:

prototype(*address*)
Returns a foreign function at the specified address which must be an integer.

prototype(*callable*)
Create a C callable function (a callback function) from a Python *callable*.

prototype(*func_spec*[, *paramflags*])
Returns a foreign function exported by a shared library. *func_spec* must be a 2-tuple `(name_or_ordinal, library)`. The first item is the name of the exported function as string, or the ordinal of the exported function as small integer. The second item is the shared library instance.

prototype(*vtbl_index*, *name*[, *paramflags*[, *iid*]])
Returns a foreign function that will call a COM method. *vtbl_index* is the index into the virtual function table, a small non-negative integer. *name* is name of the COM method. *iid* is an optional pointer to the interface identifier which is used in extended error reporting.

COM methods use a special calling convention: They require a pointer to the COM interface as first argument, in addition to those parameters that are specified in the `argtypes` tuple.

The optional *paramflags* parameter creates foreign function wrappers with much more functionality than the features described above.

paramflags must be a tuple of the same length as `argtypes`.

Each item in this tuple contains further information about a parameter, it must be a tuple containing one, two, or three items.

The first item is an integer containing a combination of direction flags for the parameter:

1 Specifies an input parameter to the function.

2 Output parameter. The foreign function fills in a value.

4 Input parameter which defaults to the integer zero.

The optional second item is the parameter name as string. If this is specified, the foreign function can be called with named parameters.

The optional third item is the default value for this parameter.

This example demonstrates how to wrap the Windows `MessageBoxW` function so that it supports default parameters and named arguments. The C declaration from the windows header file is this:

```
WINUSERAPI int WINAPI
MessageBoxW(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType);
```

Here is the wrapping with *ctypes*:

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCWSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCWSTR, LPCWSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", "Hello from ctypes
↪"), (1, "flags", 0)
>>> MessageBox = prototype(("MessageBoxW", windll.user32), paramflags)
```

The `MessageBox` foreign function can now be called in these ways:

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
```

A second example demonstrates output parameters. The win32 `GetWindowRect` function retrieves the dimensions of a specified window by copying them into `RECT` structure that the caller has to supply. Here is the C declaration:

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
     HWND hWnd,
     LPRECT lpRect);
```

Here is the wrapping with *ctypes*:

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)
>>>
```

Functions with output parameters will automatically return the output parameter value if there is a single one, or a tuple containing the output parameter values when there are more than one, so the GetWindowRect function now returns a RECT instance, when called.

Output parameters can be combined with the `errcheck` protocol to do further output processing and error checking. The win32 `GetWindowRect` api function returns a `BOOL` to signal success or failure, so this function could do the error checking, and raises an exception when the api call failed:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

If the `errcheck` function returns the argument tuple it receives unchanged, *ctypes* continues the normal processing it does on the output parameters. If you want to return a tuple of window coordinates instead of a `RECT` instance, you can retrieve the fields in the function and return them instead, the normal processing will no longer take place:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

Funções utilitárias

`ctypes.`**`addressof`**`(`*obj*`)`
Returns the address of the memory buffer as integer. *obj* must be an instance of a ctypes type.

`ctypes.`**`alignment`**`(`*obj_or_type*`)`
Returns the alignment requirements of a ctypes type. *obj_or_type* must be a ctypes type or instance.

`ctypes.`**`byref`**`(`*obj*[, *offset*]`)`
Returns a light-weight pointer to *obj*, which must be an instance of a ctypes type. *offset* defaults to zero, and must be an integer that will be added to the internal pointer value.

`byref(obj, offset)` corresponds to this C code:

```
(((char *)&obj) + offset)
```

The returned object can only be used as a foreign function call parameter. It behaves similar to `pointer(obj)`, but the construction is a lot faster.

`ctypes.`**`cast`**(*obj*, *type*)
This function is similar to the cast operator in C. It returns a new instance of *type* which points to the same memory block as *obj*. *type* must be a pointer type, and *obj* must be an object that can be interpreted as a pointer.

`ctypes.`**`create_string_buffer`**(*init_or_size*, *size=None*)
This function creates a mutable character buffer. The returned object is a ctypes array of *c_char*.

init_or_size must be an integer which specifies the size of the array, or a bytes object which will be used to initialize the array items.

If a bytes object is specified as first argument, the buffer is made one item larger than its length so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the bytes should not be used.

`ctypes.`**`create_unicode_buffer`**(*init_or_size*, *size=None*)
This function creates a mutable unicode character buffer. The returned object is a ctypes array of *c_wchar*.

init_or_size must be an integer which specifies the size of the array, or a string which will be used to initialize the array items.

If a string is specified as first argument, the buffer is made one item larger than the length of the string so that the last element in the array is a NUL termination character. An integer can be passed as second argument which allows specifying the size of the array if the length of the string should not be used.

`ctypes.`**`DllCanUnloadNow`**()
Windows only: This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllCanUnloadNow function that the _ctypes extension dll exports.

`ctypes.`**`DllGetClassObject`**()
Windows only: This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllGetClassObject function that the `_ctypes` extension dll exports.

`ctypes.util.`**`find_library`**(*name*)
Try to find a library and return a pathname. *name* is the library name without any prefix like `lib`, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option `-l`). If no library can be found, returns `None`.

The exact functionality is system dependent.

`ctypes.util.`**`find_msvcrt`**()
Windows only: return the filename of the VC runtime library used by Python, and by the extension modules. If the name of the library cannot be determined, `None` is returned.

If you need to free memory, for example, allocated by an extension module with a call to the `free(void *)`, it is important that you use the function in the same library that allocated the memory.

`ctypes.`**`FormatError`**([*code*])
Windows only: Returns a textual description of the error code *code*. If no error code is specified, the last error code is used by calling the Windows api function GetLastError.

`ctypes.`**`GetLastError`**()
Windows only: Returns the last error code set by Windows in the calling thread. This function calls the Windows *GetLastError()* function directly, it does not return the ctypes-private copy of the error code.

ctypes.**get_errno**()
Returns the current value of the ctypes-private copy of the system *errno* variable in the calling thread.

ctypes.**get_last_error**()
Windows only: returns the current value of the ctypes-private copy of the system `LastError` variable in the calling thread.

ctypes.**memmove**(*dst*, *src*, *count*)
Same as the standard C memmove library function: copies *count* bytes from *src* to *dst*. *dst* and *src* must be integers or ctypes instances that can be converted to pointers.

ctypes.**memset**(*dst*, *c*, *count*)
Same as the standard C memset library function: fills the memory block at address *dst* with *count* bytes of value *c*. *dst* must be an integer specifying an address, or a ctypes instance.

ctypes.**POINTER**(*type*)
This factory function creates and returns a new ctypes pointer type. Pointer types are cached and reused internally, so calling this function repeatedly is cheap. *type* must be a ctypes type.

ctypes.**pointer**(*obj*)
This function creates a new pointer instance, pointing to *obj*. The returned object is of the type `POINTER(type(obj))`.

Note: If you just want to pass a pointer to an object to a foreign function call, you should use `byref(obj)` which is much faster.

ctypes.**resize**(*obj*, *size*)
This function resizes the internal memory buffer of *obj*, which must be an instance of a ctypes type. It is not possible to make the buffer smaller than the native size of the objects type, as given by `sizeof(type(obj))`, but it is possible to enlarge the buffer.

ctypes.**set_errno**(*value*)
Set the current value of the ctypes-private copy of the system *errno* variable in the calling thread to *value* and return the previous value.

ctypes.**set_last_error**(*value*)
Windows only: set the current value of the ctypes-private copy of the system `LastError` variable in the calling thread to *value* and return the previous value.

ctypes.**sizeof**(*obj_or_type*)
Returns the size in bytes of a ctypes type or instance memory buffer. Does the same as the C `sizeof` operator.

ctypes.**string_at**(*address*, *size=-1*)
This function returns the C string starting at memory address *address* as a bytes object. If size is specified, it is used as size, otherwise the string is assumed to be zero-terminated.

ctypes.**WinError**(*code=None*, *descr=None*)
Windows only: this function is probably the worst-named thing in ctypes. It creates an instance of OSError. If *code* is not specified, `GetLastError` is called to determine the error code. If *descr* is not specified, *FormatError()* is called to get a textual description of the error.

Alterado na versão 3.3: An instance of *WindowsError* used to be created.

ctypes.**wstring_at**(*address*, *size=-1*)
This function returns the wide character string starting at memory address *address* as a string. If *size* is specified, it is used as the number of characters of the string, otherwise the string is assumed to be zero-terminated.

Data types

class ctypes.**_CData**

This non-public class is the common base class of all ctypes data types. Among other things, all ctypes type instances contain a memory block that hold C compatible data; the address of the memory block is returned by the *addressof()* helper function. Another instance variable is exposed as *_objects*; this contains other Python objects that need to be kept alive in case the memory block contains pointers.

Common methods of ctypes data types, these are all class methods (to be exact, they are methods of the *metaclass*):

from_buffer(*source*[, *offset*])

This method returns a ctypes instance that shares the buffer of the *source* object. The *source* object must support the writeable buffer interface. The optional *offset* parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a *ValueError* is raised.

from_buffer_copy(*source*[, *offset*])

This method creates a ctypes instance, copying the buffer from the *source* object buffer which must be readable. The optional *offset* parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a *ValueError* is raised.

from_address(*address*)

This method returns a ctypes type instance using the memory specified by *address* which must be an integer.

from_param(*obj*)

This method adapts *obj* to a ctypes type. It is called with the actual object used in a foreign function call when the type is present in the foreign function's `argtypes` tuple; it must return an object that can be used as a function call parameter.

All ctypes data types have a default implementation of this classmethod that normally returns *obj* if that is an instance of the type. Some types accept other objects as well.

in_dll(*library*, *name*)

This method returns a ctypes type instance exported by a shared library. *name* is the name of the symbol that exports the data, *library* is the loaded shared library.

Common instance variables of ctypes data types:

_b_base_

Sometimes ctypes data instances do not own the memory block they contain, instead they share part of the memory block of a base object. The *_b_base_* read-only member is the root ctypes object that owns the memory block.

_b_needsfree_

This read-only variable is true when the ctypes data instance has allocated the memory block itself, false otherwise.

_objects

This member is either `None` or a dictionary containing Python objects that need to be kept alive so that the memory block contents is kept valid. This object is only exposed for debugging; never modify the contents of this dictionary.

Fundamental data types

class ctypes.**_SimpleCData**
This non-public class is the base class of all fundamental ctypes data types. It is mentioned here because it contains the common attributes of the fundamental ctypes data types. *_SimpleCData* is a subclass of *_CData*, so it inherits their methods and attributes. ctypes data types that are not and do not contain pointers can now be pickled.

Instances have a single attribute:

value
This attribute contains the actual value of the instance. For integer and pointer types, it is an integer, for character types, it is a single character bytes object or string, for character pointer types it is a Python bytes object or string.

When the `value` attribute is retrieved from a ctypes instance, usually a new object is returned each time. *ctypes* does *not* implement original object return, always a new object is constructed. The same is true for all other ctypes object instances.

Fundamental data types, when returned as foreign function call results, or, for example, by retrieving structure field members or array items, are transparently converted to native Python types. In other words, if a foreign function has a `restype` of *c_char_p*, you will always receive a Python bytes object, *not* a *c_char_p* instance.

Subclasses of fundamental data types do *not* inherit this behavior. So, if a foreign functions `restype` is a subclass of *c_void_p*, you will receive an instance of this subclass from the function call. Of course, you can get the value of the pointer by accessing the `value` attribute.

These are the fundamental ctypes data types:

class ctypes.**c_byte**
Represents the C `signed char` datatype, and interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_char**
Represents the C `char` datatype, and interprets the value as a single character. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.**c_char_p**
Represents the C `char *` datatype when it points to a zero-terminated string. For a general character pointer that may also point to binary data, `POINTER(c_char)` must be used. The constructor accepts an integer address, or a bytes object.

class ctypes.**c_double**
Represents the C `double` datatype. The constructor accepts an optional float initializer.

class ctypes.**c_longdouble**
Represents the C `long double` datatype. The constructor accepts an optional float initializer. On platforms where `sizeof(long double) == sizeof(double)` it is an alias to *c_double*.

class ctypes.**c_float**
Represents the C `float` datatype. The constructor accepts an optional float initializer.

class ctypes.**c_int**
Represents the C `signed int` datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias to *c_long*.

class ctypes.**c_int8**
Represents the C 8-bit `signed int` datatype. Usually an alias for *c_byte*.

class ctypes.**c_int16**
Represents the C 16-bit `signed int` datatype. Usually an alias for *c_short*.

class ctypes.**c_int32**
Represents the C 32-bit `signed int` datatype. Usually an alias for *c_int*.

class ctypes.**c_int64**
Represents the C 64-bit `signed int` datatype. Usually an alias for *c_longlong*.

class ctypes.**c_long**
Represents the C `signed long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_longlong**
Represents the C `signed long long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_short**
Represents the C `signed short` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_size_t**
Represents the C `size_t` datatype.

class ctypes.**c_ssize_t**
Represents the C `ssize_t` datatype.

Novo na versão 3.2.

class ctypes.**c_ubyte**
Represents the C `unsigned char` datatype, it interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_uint**
Represents the C `unsigned int` datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias for *c_ulong*.

class ctypes.**c_uint8**
Represents the C 8-bit `unsigned int` datatype. Usually an alias for *c_ubyte*.

class ctypes.**c_uint16**
Represents the C 16-bit `unsigned int` datatype. Usually an alias for *c_ushort*.

class ctypes.**c_uint32**
Represents the C 32-bit `unsigned int` datatype. Usually an alias for *c_uint*.

class ctypes.**c_uint64**
Represents the C 64-bit `unsigned int` datatype. Usually an alias for *c_ulonglong*.

class ctypes.**c_ulong**
Represents the C `unsigned long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_ulonglong**
Represents the C `unsigned long long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_ushort**
Represents the C `unsigned short` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_void_p**
Represents the C `void *` type. The value is represented as integer. The constructor accepts an optional integer initializer.

class ctypes.**c_wchar**
Represents the C `wchar_t` datatype, and interprets the value as a single character unicode string. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.**c_wchar_p**
Represents the C `wchar_t *` datatype, which must be a pointer to a zero-terminated wide character string. The constructor accepts an integer address, or a string.

class ctypes.**c_bool**
Represent the C `bool` datatype (more accurately, `_Bool` from C99). Its value can be `True` or `False`, and the constructor accepts any object that has a truth value.

class ctypes.**HRESULT**
Windows only: Represents a `HRESULT` value, which contains success or error information for a function or method call.

class ctypes.**py_object**
Represents the C `PyObject *` datatype. Calling this without an argument creates a `NULL` `PyObject *` pointer.

The `ctypes.wintypes` module provides quite some other Windows specific data types, for example `HWND`, `WPARAM`, or `DWORD`. Some useful structures like `MSG` or `RECT` are also defined.

Structured data types

class ctypes.**Union**(**args*, ***kw*)
Abstract base class for unions in native byte order.

class ctypes.**BigEndianStructure**(**args*, ***kw*)
Abstract base class for structures in *big endian* byte order.

class ctypes.**LittleEndianStructure**(**args*, ***kw*)
Abstract base class for structures in *little endian* byte order.

Structures with non-native byte order cannot contain pointer type fields, or any other data types containing pointer type fields.

class ctypes.**Structure**(**args*, ***kw*)
Abstract base class for structures in *native* byte order.

Concrete structure and union types must be created by subclassing one of these types, and at least define a *_fields_* class variable. *ctypes* will create *descriptor*s which allow reading and writing the fields by direct attribute accesses. These are the

fields
A sequence defining the structure fields. The items must be 2-tuples or 3-tuples. The first item is the name of the field, the second item specifies the type of the field; it can be any ctypes data type.

For integer type fields like *c_int*, a third optional item can be given. It must be a small positive integer defining the bit width of the field.

Field names must be unique within one structure or union. This is not checked, only one field can be accessed when names are repeated.

It is possible to define the *_fields_* class variable *after* the class statement that defines the Structure subclass, this allows creating data types that directly or indirectly reference themselves:

```
class List(Structure):
    pass
List._fields_ = [("pnext", POINTER(List)),
```

(continua na próxima página)

(continuação da página anterior)

```
                ...
               ]
```

The *_fields_* class variable must, however, be defined before the type is first used (an instance is created, *sizeof()* is called on it, and so on). Later assignments to the *_fields_* class variable will raise an AttributeError.

It is possible to define sub-subclasses of structure types, they inherit the fields of the base class plus the *_fields_* defined in the sub-subclass, if any.

pack

An optional small integer that allows overriding the alignment of structure fields in the instance. *_pack_* must already be defined when *_fields_* is assigned, otherwise it will have no effect.

anonymous

An optional sequence that lists the names of unnamed (anonymous) fields. *_anonymous_* must be already defined when *_fields_* is assigned, otherwise it will have no effect.

The fields listed in this variable must be structure or union type fields. *ctypes* will create descriptors in the structure type that allows accessing the nested fields directly, without the need to create the structure or union field.

Here is an example type (Windows):

```
class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]
```

The `TYPEDESC` structure describes a COM data type, the `vt` field specifies which one of the union fields is valid. Since the `u` field is defined as anonymous field, it is now possible to access the members directly off the TYPEDESC instance. `td.lptdesc` and `td.u.lptdesc` are equivalent, but the former is faster since it does not need to create a temporary union instance:

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

It is possible to define sub-subclasses of structures, they inherit the fields of the base class. If the subclass definition has a separate *_fields_* variable, the fields specified in this are appended to the fields of the base class.

Structure and union constructors accept both positional and keyword arguments. Positional arguments are used to initialize member fields in the same order as they are appear in *_fields_*. Keyword arguments in the constructor are interpreted as attribute assignments, so they will initialize *_fields_* with the same name, or create new attributes for names not present in *_fields_*.

Arrays and pointers

class ctypes.**Array**(**args*)
Abstract base class for arrays.

The recommended way to create concrete array types is by multiplying any *ctypes* data type with a positive integer. Alternatively, you can subclass this type and define *_length_* and *_type_* class variables. Array elements can be read and written using standard subscript and slice accesses; for slice reads, the resulting object is *not* itself an *Array*.

length
A positive integer specifying the number of elements in the array. Out-of-range subscripts result in an *IndexError*. Will be returned by *len()*.

type
Specifies the type of each element in the array.

Array subclass constructors accept positional arguments, used to initialize the elements in order.

class ctypes.**_Pointer**
Private, abstract base class for pointers.

Concrete pointer types are created by calling *POINTER()* with the type that will be pointed to; this is done automatically by *pointer()*.

If a pointer points to an array, its elements can be read and written using standard subscript and slice accesses. Pointer objects have no size, so *len()* will raise *TypeError*. Negative subscripts will read from the memory *before* the pointer (as in C), and out-of-range subscripts will probably crash with an access violation (if you're lucky).

type
Specifies the type pointed to.

contents
Returns the object to which to pointer points. Assigning to this attribute changes the pointer to point to the assigned object.

CAPÍTULO 17

Execução Concorrente

Os módulos descritos neste capítulo fornecem suporte a execução simultânea de código. A escolha apropriada da ferramenta dependerá da tarefa a ser executada (CPU bound ou IO bound) e do estilo de desenvolvimento preferencial (multitarefa cooperativa orientada a eventos versus multitarefa preemptiva). Eis uma visão geral:

17.1 `threading` — Paralelismo baseado em Thread

Código Fonte: Lib/threading.py

Este módulo constrói interfaces de alto nível para threading usando o módulo *`_thread`*, de mais baixo nível. Veja também o módulo *`queue`*.

Alterado na versão 3.7: Este módulo costumava ser opcional, agora está sempre disponível.

Nota: Mesmo não sendo listadas abaixo, os nomes `camelCase` usados por alguns métodos e funções neste módulo, no Python 2.x, ainda são suportados pelo módulo.

CPython implementation detail: In CPython, due to the *Global Interpreter Lock*, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use *`multiprocessing`* or *`concurrent.futures.ProcessPoolExecutor`*. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

Este módulo define as seguintes funções:

`threading.`**`active_count`**`()`
Retorna o número de objetos *`Thread`* atualmente ativos. A quantidade retornada é igual ao tamanho da lista retornada por *`enumerate()`*.

`threading.`**`current_thread`**`()`
Retorna o objeto *`Thread`* atual, correspondendo ao controle do chamador da thread. Se o controle do chamador

da thread não foi criado através do módulo *threading*, um objeto thread vazio, com funcionalidade limitada, é retornado.

threading.**get_ident**()
Retorna o 'identificador de thread' do thread atual. Este é um número inteiro diferente de zero. Seu valor não tem significado direto; pretende-se que seja um cookie mágico para ser usado, por exemplo, para indexar um dicionário de dados específicos do thread. identificadores de thread podem ser reciclados quando um thread sai e outro é criado.

Novo na versão 3.3.

threading.**enumerate**()
Return a list of all *Thread* objects currently alive. The list includes daemonic threads, dummy thread objects created by *current_thread()*, and the main thread. It excludes terminated threads and threads that have not yet been started.

threading.**main_thread**()
Return the main *Thread* object. In normal conditions, the main thread is the thread from which the Python interpreter was started.

Novo na versão 3.4.

threading.**settrace**(*func*)
Set a trace function for all threads started from the *threading* module. The *func* will be passed to *sys.settrace()* for each thread, before its *run()* method is called.

threading.**setprofile**(*func*)
Set a profile function for all threads started from the *threading* module. The *func* will be passed to *sys.setprofile()* for each thread, before its *run()* method is called.

threading.**stack_size**([*size*])
Retorna o tamanho da pilha de threads usado ao criar novos threads. O argumento opcional *size* especifica o tamanho da pilha a ser usado para threads criados posteriormente e deve ser 0 (usar plataforma ou padrão configurado) ou um valor inteiro positivo de pelo menos 32.768 (32 KiB). Se *size* não for especificado, 0 será usado. Se a alteração do tamanho da pilha de threads não for suportada, um :exc: *RuntimeError* será gerado. Se o tamanho da pilha especificado for inválido, um :exc: *ValueError* será aumentado e o tamanho da pilha não será modificado. Atualmente, 0 KiB é o valor mínimo de tamanho de pilha suportado para garantir espaço suficiente para o próprio intérprete. Observe que algumas plataformas podem ter restrições específicas sobre valores para o tamanho da pilha, como exigir um tamanho mínimo de pilha> 32 KiB ou exigir alocação em múltiplos do tamanho da página de memória do sistema - a documentação da plataforma deve ser consultada para obter mais informações (4 páginas KiB são comuns; usar múltiplos de 4096 para o tamanho da pilha é a abordagem sugerida na ausência de informações mais específicas).

Availability: Windows, systems with POSIX threads.

This module also defines the following constant:

threading.**TIMEOUT_MAX**
The maximum value allowed for the *timeout* parameter of blocking functions (*Lock.acquire()*, *RLock.acquire()*, *Condition.wait()*, etc.). Specifying a timeout greater than this value will raise an *OverflowError*.

Novo na versão 3.2.

This module defines a number of classes, which are detailed in the sections below.

The design of this module is loosely based on Java's threading model. However, where Java makes locks and condition variables basic behavior of every object, they are separate objects in Python. Python's *Thread* class supports a subset of the behavior of Java's Thread class; currently, there are no priorities, no thread groups, and threads cannot be destroyed, stopped, suspended, resumed, or interrupted. The static methods of Java's Thread class, when implemented, are mapped to module-level functions.

All of the methods described below are executed atomically.

17.1.1 Thread-Local Data

Thread-local data is data whose values are thread specific. To manage thread-local data, just create an instance of *local* (or a subclass) and store attributes on it:

```
mydata = threading.local()
mydata.x = 1
```

The instance's values will be different for separate threads.

class threading.**local**
A class that represents thread-local data.

For more details and extensive examples, see the documentation string of the _threading_local module.

17.1.2 Thread Objects

The *Thread* class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the *run()* method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, *only* override the __init__() and *run()* methods of this class.

Once a thread object is created, its activity must be started by calling the thread's *start()* method. This invokes the *run()* method in a separate thread of control.

Once the thread's activity is started, the thread is considered 'alive'. It stops being alive when its *run()* method terminates – either normally, or by raising an unhandled exception. The *is_alive()* method tests whether the thread is alive.

Other threads can call a thread's *join()* method. This blocks the calling thread until the thread whose *join()* method is called is terminated.

A thread has a name. The name can be passed to the constructor, and read or changed through the *name* attribute.

A thread can be flagged as a "daemon thread". The significance of this flag is that the entire Python program exits when only daemon threads are left. The initial value is inherited from the creating thread. The flag can be set through the *daemon* property or the *daemon* constructor argument.

Nota: Daemon threads are abruptly stopped at shutdown. Their resources (such as open files, database transactions, etc.) may not be released properly. If you want your threads to stop gracefully, make them non-daemonic and use a suitable signalling mechanism such as an *Event*.

There is a "main thread" object; this corresponds to the initial thread of control in the Python program. It is not a daemon thread.

There is the possibility that "dummy thread objects" are created. These are thread objects corresponding to "alien threads", which are threads of control started outside the threading module, such as directly from C code. Dummy thread objects have limited functionality; they are always considered alive and daemonic, and cannot be *join()*ed. They are never deleted, since it is impossible to detect the termination of alien threads.

class threading.**Thread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, ***, *daemon=None*)
This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the *run()* method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-*N*" where *N* is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If not None, *daemon* explicitly sets whether the thread is daemonic. If None (the default), the daemonic property is inherited from the current thread.

If the subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Alterado na versão 3.3: Adicionado o argumento *daemon*.

start()
Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's *run()* method to be invoked in a separate thread of control.

This method will raise a *RuntimeError* if called more than once on the same thread object.

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard *run()* method invokes the callable object passed to the object's constructor as the *target* argument, if any, with positional and keyword arguments taken from the *args* and *kwargs* arguments, respectively.

join(*timeout=None*)
Wait until the thread terminates. This blocks the calling thread until the thread whose *join()* method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs.

When the *timeout* argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As *join()* always returns None, you must call *is_alive()* after *join()* to decide whether a timeout happened – if the thread is still alive, the *join()* call timed out.

When the *timeout* argument is not present or None, the operation will block until the thread terminates.

A thread can be *join()*ed many times.

join() raises a *RuntimeError* if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to *join()* a thread before it has been started and attempts to do so raise the same exception.

name
A string used for identification purposes only. It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

getName()
setName()
Old getter/setter API for *name*; use it directly as a property instead.

ident
The 'thread identifier' of this thread or None if the thread has not been started. This is a nonzero integer. See the *get_ident()* function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

is_alive()
Return whether the thread is alive.

This method returns `True` just before the *`run()`* method starts until just after the *`run()`* method terminates. The module function *`enumerate()`* returns a list of all alive threads.

daemon
A boolean value indicating whether this thread is a daemon thread (True) or not (False). This must be set before *`start()`* is called, otherwise *`RuntimeError`* is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to *`daemon`* = `False`.

The entire Python program exits when no alive non-daemon threads are left.

isDaemon()
setDaemon()
Old getter/setter API for *`daemon`*; use it directly as a property instead.

17.1.3 Lock Objects

A primitive lock is a synchronization primitive that is not owned by a particular thread when locked. In Python, it is currently the lowest level synchronization primitive available, implemented directly by the *`_thread`* extension module.

A primitive lock is in one of two states, "locked" or "unlocked". It is created in the unlocked state. It has two basic methods, *`acquire()`* and *`release()`*. When the state is unlocked, *`acquire()`* changes the state to locked and returns immediately. When the state is locked, *`acquire()`* blocks until a call to *`release()`* in another thread changes it to unlocked, then the *`acquire()`* call resets it to locked and returns. The *`release()`* method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a *`RuntimeError`* will be raised.

Locks also support the *context management protocol*.

When more than one thread is blocked in *`acquire()`* waiting for the state to turn to unlocked, only one thread proceeds when a *`release()`* call resets the state to unlocked; which one of the waiting threads proceeds is not defined, and may vary across implementations.

All methods are executed atomically.

class threading.**Lock**
The class implementing primitive lock objects. Once a thread has acquired a lock, subsequent attempts to acquire it block, until it is released; any thread may release it.

Note that `Lock` is actually a factory function which returns an instance of the most efficient version of the concrete Lock class that is supported by the platform.

acquire(*blocking=True*, *timeout=-1*)
Acquire a lock, blocking or non-blocking.

When invoked with the *blocking* argument set to `True` (the default), block until the lock is unlocked, then set it to locked and return `True`.

When invoked with the *blocking* argument set to `False`, do not block. If a call with *blocking* set to `True` would block, return `False` immediately; otherwise, set the lock to locked and return `True`.

When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired. A *timeout* argument of `-1` specifies an unbounded wait. It is forbidden to specify a *timeout* when *blocking* is false.

The return value is `True` if the lock is acquired successfully, `False` if not (for example if the *timeout* expired).

Alterado na versão 3.2: O parâmetro *timeout* é novo.

Alterado na versão 3.2: Lock acquisition can now be interrupted by signals on POSIX if the underlying threading implementation supports it.

release()
Release a lock. This can be called from any thread, not only the thread which has acquired the lock.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

When invoked on an unlocked lock, a *RuntimeError* is raised.

There is no return value.

locked()
Return true if the lock is acquired.

17.1.4 Objetos RLock

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it uses the concepts of "owning thread" and "recursion level" in addition to the locked/unlocked state used by primitive locks. In the locked state, some thread owns the lock; in the unlocked state, no thread owns it.

To lock the lock, a thread calls its *acquire()* method; this returns once the thread owns the lock. To unlock the lock, a thread calls its *release()* method. *acquire()*/*release()* call pairs may be nested; only the final *release()* (the *release()* of the outermost pair) resets the lock to unlocked and allows another thread blocked in *acquire()* to proceed.

Reentrant locks also support the *context management protocol*.

class threading.**RLock**
This class implements reentrant lock objects. A reentrant lock must be released by the thread that acquired it. Once a thread has acquired a reentrant lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

Note that RLock is actually a factory function which returns an instance of the most efficient version of the concrete RLock class that is supported by the platform.

acquire(*blocking=True*, *timeout=-1*)
Acquire a lock, blocking or non-blocking.

When invoked without arguments: if this thread already owns the lock, increment the recursion level by one, and return immediately. Otherwise, if another thread owns the lock, block until the lock is unlocked. Once the lock is unlocked (not owned by any thread), then grab ownership, set the recursion level to one, and return. If more than one thread is blocked waiting until the lock is unlocked, only one at a time will be able to grab ownership of the lock. There is no return value in this case.

When invoked with the *blocking* argument set to true, do the same thing as when called without arguments, and return True.

When invoked with the *blocking* argument set to false, do not block. If a call without an argument would block, return False immediately; otherwise, do the same thing as when called without arguments, and return True.

When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired. Return True if the lock has been acquired, false if the timeout has elapsed.

Alterado na versão 3.2: O parâmetro *timeout* é novo.

release()
Release a lock, decrementing the recursion level. If after the decrement it is zero, reset the lock to unlocked

(not owned by any thread), and if any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling thread.

Only call this method when the calling thread owns the lock. A *RuntimeError* is raised if this method is called when the lock is unlocked.

There is no return value.

17.1.5 Condition Objects

A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default. Passing one in is useful when several condition variables must share the same lock. The lock is part of the condition object: you don't have to track it separately.

A condition variable obeys the *context management protocol*: using the `with` statement acquires the associated lock for the duration of the enclosed block. The *acquire()* and *release()* methods also call the corresponding methods of the associated lock.

Other methods must be called with the associated lock held. The *wait()* method releases the lock, and then blocks until another thread awakens it by calling *notify()* or *notify_all()*. Once awakened, *wait()* re-acquires the lock and returns. It is also possible to specify a timeout.

The *notify()* method wakes up one of the threads waiting for the condition variable, if any are waiting. The *notify_all()* method wakes up all threads waiting for the condition variable.

Note: the *notify()* and *notify_all()* methods don't release the lock; this means that the thread or threads awakened will not return from their *wait()* call immediately, but only when the thread that called *notify()* or *notify_all()* finally relinquishes ownership of the lock.

The typical programming style using condition variables uses the lock to synchronize access to some shared state; threads that are interested in a particular change of state call *wait()* repeatedly until they see the desired state, while threads that modify the state call *notify()* or *notify_all()* when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer-consumer situation with unlimited buffer capacity:

```
# Consume one item
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# Produce one item
with cv:
    make_an_item_available()
    cv.notify()
```

The `while` loop checking for the application's condition is necessary because *wait()* can return after an arbitrary long time, and the condition which prompted the *notify()* call may no longer hold true. This is inherent to multi-threaded programming. The *wait_for()* method can be used to automate the condition checking, and eases the computation of timeouts:

```
# Consume an item
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

To choose between *notify()* and *notify_all()*, consider whether one state change can be interesting for only one or several waiting threads. E.g. in a typical producer-consumer situation, adding one item to the buffer only needs to wake up one consumer thread.

class threading.**Condition**(*lock=None*)

This class implements condition variable objects. A condition variable allows one or more threads to wait until they are notified by another thread.

If the *lock* argument is given and not None, it must be a *Lock* or *RLock* object, and it is used as the underlying lock. Otherwise, a new *RLock* object is created and used as the underlying lock.

Alterado na versão 3.3: changed from a factory function to a class.

acquire(**args*)

Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.

release()

Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.

wait(*timeout=None*)

Wait until notified or until a timeout occurs. If the calling thread has not acquired the lock when this method is called, a *RuntimeError* is raised.

This method releases the underlying lock, and then blocks until it is awakened by a *notify()* or *notify_all()* call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.

When the *timeout* argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

When the underlying lock is an *RLock*, it is not released using its *release()* method, since this may not actually unlock the lock when it was acquired multiple times recursively. Instead, an internal interface of the *RLock* class is used, which really unlocks it even when it has been recursively acquired several times. Another internal interface is then used to restore the recursion level when the lock is reacquired.

The return value is True unless a given *timeout* expired, in which case it is False.

Alterado na versão 3.2: Previously, the method always returned None.

wait_for(*predicate*, *timeout=None*)

Wait until a condition evaluates to true. *predicate* should be a callable which result will be interpreted as a boolean value. A *timeout* may be provided giving the maximum time to wait.

This utility method may call *wait()* repeatedly until the predicate is satisfied, or until a timeout occurs. The return value is the last return value of the predicate and will evaluate to False if the method timed out.

Ignoring the timeout feature, calling this method is roughly equivalent to writing:

```
while not predicate():
    cv.wait()
```

Therefore, the same rules apply as with *wait()*: The lock must be held when called and is re-acquired on return. The predicate is evaluated with the lock held.

Novo na versão 3.2.

notify(*n=1*)

By default, wake up one thread waiting on this condition, if any. If the calling thread has not acquired the lock when this method is called, a *RuntimeError* is raised.

This method wakes up at most *n* of the threads waiting for the condition variable; it is a no-op if no threads are waiting.

The current implementation wakes up exactly *n* threads, if at least *n* threads are waiting. However, it's not safe to rely on this behavior. A future, optimized implementation may occasionally wake up more than *n* threads.

Note: an awakened thread does not actually return from its `wait()` call until it can reacquire the lock. Since `notify()` does not release the lock, its caller should.

notify_all()
Wake up all threads waiting on this condition. This method acts like `notify()`, but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

17.1.6 Semaphore Objects

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra (he used the names `P()` and `V()` instead of `acquire()` and `release()`).

A semaphore manages an internal counter which is decremented by each `acquire()` call and incremented by each `release()` call. The counter can never go below zero; when `acquire()` finds that it is zero, it blocks, waiting until some other thread calls `release()`.

Semaphores also support the *context management protocol*.

class threading.**Semaphore**(*value=1*)
This class implements semaphore objects. A semaphore manages an atomic counter representing the number of `release()` calls minus the number of `acquire()` calls, plus an initial value. The `acquire()` method blocks if necessary until it can return without making the counter negative. If not given, *value* defaults to 1.

The optional argument gives the initial *value* for the internal counter; it defaults to `1`. If the *value* given is less than 0, `ValueError` is raised.

Alterado na versão 3.3: changed from a factory function to a class.

acquire(*blocking=True*, *timeout=None*)
Acquire a semaphore.

When invoked without arguments:

- If the internal counter is larger than zero on entry, decrement it by one and return `True` immediately.
- If the internal counter is zero on entry, block until awoken by a call to `release()`. Once awoken (and the counter is greater than 0), decrement the counter by 1 and return `True`. Exactly one thread will be awoken by each call to `release()`. The order in which threads are awoken should not be relied on.

When invoked with *blocking* set to false, do not block. If a call without an argument would block, return `False` immediately; otherwise, do the same thing as when called without arguments, and return `True`.

When invoked with a *timeout* other than `None`, it will block for at most *timeout* seconds. If acquire does not complete successfully in that interval, return `False`. Return `True` otherwise.

Alterado na versão 3.2: O parâmetro *timeout* é novo.

release()
Release a semaphore, incrementing the internal counter by one. When it was zero on entry and another thread is waiting for it to become larger than zero again, wake up that thread.

class threading.**BoundedSemaphore**(*value=1*)
Class implementing bounded semaphore objects. A bounded semaphore checks to make sure its current value doesn't exceed its initial value. If it does, `ValueError` is raised. In most situations semaphores are used to guard

resources with limited capacity. If the semaphore is released too many times it's a sign of a bug. If not given, *value* defaults to 1.

Alterado na versão 3.3: changed from a factory function to a class.

Exemplo `Semaphore`

Semaphores are often used to guard resources with limited capacity, for example, a database server. In any situation where the size of the resource is fixed, you should use a bounded semaphore. Before spawning any worker threads, your main thread would initialize the semaphore:

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

Once spawned, worker threads call the semaphore's acquire and release methods when they need to connect to the server:

```
with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
    finally:
        conn.close()
```

The use of a bounded semaphore reduces the chance that a programming error which causes the semaphore to be released more than it's acquired will go undetected.

17.1.7 Event Objects

This is one of the simplest mechanisms for communication between threads: one thread signals an event and other threads wait for it.

An event object manages an internal flag that can be set to true with the *set()* method and reset to false with the *clear()* method. The *wait()* method blocks until the flag is true.

class threading.**Event**

Class implementing event objects. An event manages a flag that can be set to true with the *set()* method and reset to false with the *clear()* method. The *wait()* method blocks until the flag is true. The flag is initially false.

Alterado na versão 3.3: changed from a factory function to a class.

is_set()

Return `True` if and only if the internal flag is true.

set()

Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call *wait()* once the flag is true will not block at all.

clear()

Reset the internal flag to false. Subsequently, threads calling *wait()* will block until *set()* is called to set the internal flag to true again.

wait(*timeout=None*)

Block until the internal flag is true. If the internal flag is true on entry, return immediately. Otherwise, block until another thread calls *set()* to set the flag to true, or until the optional timeout occurs.

When the timeout argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

This method returns `True` if and only if the internal flag has been set to true, either before the wait call or after the wait starts, so it will always return `True` except if a timeout is given and the operation times out.

Alterado na versão 3.1: Previously, the method always returned `None`.

17.1.8 Objetos Timer

This class represents an action that should be run only after a certain amount of time has passed — a timer. *Timer* is a subclass of *Thread* and as such also functions as an example of creating custom threads.

Timers are started, as with threads, by calling their `start()` method. The timer can be stopped (before its action has begun) by calling the *cancel()* method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

Por exemplo:

```
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

class threading.**Timer**(*interval*, *function*, *args=None*, *kwargs=None*)

Create a timer that will run *function* with arguments *args* and keyword arguments *kwargs*, after *interval* seconds have passed. If *args* is `None` (the default) then an empty list will be used. If *kwargs* is `None` (the default) then an empty dict will be used.

Alterado na versão 3.3: changed from a factory function to a class.

cancel()

Stop the timer, and cancel the execution of the timer's action. This will only work if the timer is still in its waiting stage.

17.1.9 Barrier Objects

Novo na versão 3.2.

This class provides a simple synchronization primitive for use by a fixed number of threads that need to wait for each other. Each of the threads tries to pass the barrier by calling the *wait()* method and will block until all of the threads have made their *wait()* calls. At this point, the threads are released simultaneously.

The barrier can be reused any number of times for the same number of threads.

As an example, here is a simple way to synchronize a client and server thread:

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)
```

(continua na próxima página)

(continuação da página anterior)

```
def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

class threading.**Barrier**(*parties*, *action=None*, *timeout=None*)

Create a barrier object for *parties* number of threads. An *action*, when provided, is a callable to be called by one of the threads when they are released. *timeout* is the default timeout value if none is specified for the *wait()* method.

wait(*timeout=None*)

Pass the barrier. When all the threads party to the barrier have called this function, they are all released simultaneously. If a *timeout* is provided, it is used in preference to any that was supplied to the class constructor.

The return value is an integer in the range 0 to *parties* – 1, different for each thread. This can be used to select a thread to do some special housekeeping, e.g.:

```
i = barrier.wait()
if i == 0:
    # Only one thread needs to print this
    print("passed the barrier")
```

If an *action* was provided to the constructor, one of the threads will have called it prior to being released. Should this call raise an error, the barrier is put into the broken state.

If the call times out, the barrier is put into the broken state.

This method may raise a *BrokenBarrierError* exception if the barrier is broken or reset while a thread is waiting.

reset()

Return the barrier to the default, empty state. Any threads waiting on it will receive the *BrokenBarrierError* exception.

Note that using this function may can require some external synchronization if there are other threads whose state is unknown. If a barrier is broken it may be better to just leave it and create a new one.

abort()

Put the barrier into a broken state. This causes any active or future calls to *wait()* to fail with the *BrokenBarrierError*. Use this for example if one of the needs to abort, to avoid deadlocking the application.

It may be preferable to simply create the barrier with a sensible *timeout* value to automatically guard against one of the threads going awry.

parties

The number of threads required to pass the barrier.

n_waiting

The number of threads currently waiting in the barrier.

broken

A boolean that is `True` if the barrier is in the broken state.

exception threading.**BrokenBarrierError**

This exception, a subclass of *RuntimeError*, is raised when the *Barrier* object is reset or broken.

17.1.10 Using locks, conditions, and semaphores in the `with` statement

All of the objects provided by this module that have `acquire()` and `release()` methods can be used as context managers for a `with` statement. The `acquire()` method will be called when the block is entered, and `release()` will be called when the block is exited. Hence, the following snippet:

```
with some_lock:
    # do something...
```

is equivalent to:

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

Currently, *Lock*, *RLock*, *Condition*, *Semaphore*, and *BoundedSemaphore* objects may be used as `with` statement context managers.

17.2 `threading` — Paralelismo baseado em processo

Código Fonte: Lib/multiprocessing/

17.2.1 Introdução

multiprocessing is a package that supports spawning processes using an API similar to the *threading* module. The *multiprocessing* package offers both local and remote concurrency, effectively side-stepping the *Global Interpreter Lock* by using subprocesses instead of threads. Due to this, the *multiprocessing* module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows.

The *multiprocessing* module also introduces APIs which do not have analogs in the *threading* module. A prime example of this is the *Pool* object which offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data parallelism). The following example demonstrates the common practice of defining such functions in a module so that child processes can successfully import that module. This basic example of data parallelism using *Pool*,

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

irá exibir na saída padrão

```
[1, 4, 9]
```

A classe `Process`

In *multiprocessing*, processes are spawned by creating a *Process* object and then calling its *start()* method. *Process* follows the API of *threading.Thread*. A trivial example of a multiprocess program is

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

Para mostrar os IDs de processo individuais envolvidos, aqui está um exemplo expandido:

```
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

Para uma explicação de porque a parte `if __name__ == '__main__'` é necessária, veja *Programming guidelines*.

Contextos e métodos de inicialização

Dependendo da plataforma, *multiprocessing* suporta três maneiras de iniciar um processo. Estes *métodos de início* são

spawn The parent process starts a fresh python interpreter process. The child process will only inherit those resources necessary to run the process objects *run()* method. In particular, unnecessary file descriptors and handles from the parent process will not be inherited. Starting a process using this method is rather slow compared to using *fork* or *forkserver*.

Available on Unix and Windows. The default on Windows.

fork The parent process uses *os.fork()* to fork the Python interpreter. The child process, when it begins, is effectively identical to the parent process. All resources of the parent are inherited by the child process. Note that safely forking a multithreaded process is problematic.

Disponível apenas para Unix. O padrão no Unix.

forkserver Quando o programa é inicializado e seleciona o método de início *forkserver*, um processo de servidor é inicializado. A partir disso, sempre que um novo processo é necessário, o processo pai

conecta-se ao servidor e solicita que um novo processo seja bifurcado. O fork do processo do servidor é de thread unico, então é seguro para utilizar *os.fork()*. Nenhum recurso desnecessário é herdado.

Disponível em plataformas Unix que suportam a passagem de descritores de arquivo em Unix pipes.

Alterado na versão 3.4: *spawn* added on all unix platforms, and *forkserver* added for some unix platforms. Child processes no longer inherit all of the parents inheritable handles on Windows.

On Unix using the *spawn* or *forkserver* start methods will also start a *semaphore tracker* process which tracks the unlinked named semaphores created by processes of the program. When all processes have exited the semaphore tracker unlinks any remaining semaphores. Usually there should be none, but if a process was killed by a signal there may be some "leaked" semaphores. (Unlinking the named semaphores is a serious matter since the system allows only a limited number, and they will not be automatically unlinked until the next reboot.)

To select a start method you use the *set_start_method()* in the `if __name__ == '__main__'` clause of the main module. For example:

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    mp.set_start_method('spawn')
    q = mp.Queue()
    p = mp.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

set_start_method() should not be used more than once in the program.

Alternatively, you can use *get_context()* to obtain a context object. Context objects have the same API as the multiprocessing module, and allow one to use multiple start methods in the same program.

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    ctx = mp.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

Note that objects related to one context may not be compatible with processes for a different context. In particular, locks created using the *fork* context cannot be passed to processes started using the *spawn* or *forkserver* start methods.

Uma biblioteca que deseja utilizar um método de início específico provavelmente deve utilizar *get_context()* para evitar interferir na escolha do usuário.

Aviso: The `'spawn'` and `'forkserver'` start methods cannot currently be used with "frozen" executables (i.e., binaries produced by packages like **PyInstaller** and **cx_Freeze**) on Unix. The `'fork'` start method does work.

Trocando objetos entre processos

multiprocessing supports two types of communication channel between processes:

Filas

A classe *Queue* é quase um clone de *queue.Queue*. Por exemplo:

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

Queues are thread and process safe.

Pipes

The *Pipe()* function returns a pair of connection objects connected by a pipe which by default is duplex (two-way). For example:

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # prints "[42, None, 'hello']"
    p.join()
```

The two connection objects returned by *Pipe()* represent the two ends of the pipe. Each connection object has `send()` and `recv()` methods (among others). Note that data in a pipe may become corrupted if two processes (or threads) try to read from or write to the *same* end of the pipe at the same time. Of course there is no risk of corruption from processes using different ends of the pipe at the same time.

Sincronização entre processos

multiprocessing contains equivalents of all the synchronization primitives from *threading*. For instance one can use a lock to ensure that only one process prints to standard output at a time:

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
```

(continua na próxima página)

(continuação da página anterior)

```
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

Sem utilizar a saída do bloqueio dos diferentes processos, é possível que tudo fique confuso.

Compartilhando estado entre processos.

Conforme mencionado acima, ao fazer programação concorrente, geralmente é melhor evitar o uso de estado compartilhado, tanto quanto possível. Isso é particularmente verdadeiro ao utilizar múltiplos processos.

No entanto, se você realmente precisa utilizar algum compartilhamento de dados, então *multiprocessing* fornece algumas maneiras de se fazer isso.

Shared memory

Os dados podem ser armazenados em um mapa de memória compartilhado utilizando *Value* ou `vetor`. Por exemplo, o código a seguir

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

será impresso:

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

The `'d'` and `'i'` arguments used when creating `num` and `arr` are typecodes of the kind used by the *array* module: `'d'` indicates a double precision float and `'i'` indicates a signed integer. These shared objects will be process and thread-safe.

Para mais flexibilidade no uso de memória compartilhada, pode-se utilizar o módulo *multiprocessing.sharedctypes*, que suporta a criação de objetos ctypes arbitrários alocados da memória compartilhada.

Processo do Servidor

A manager object returned by `Manager()` controls a server process which holds Python objects and allows other processes to manipulate them using proxies.

A manager returned by `Manager()` will support types *list*, *dict*, *Namespace*, *Lock*, *RLock*, *Semaphore*, *BoundedSemaphore*, *Condition*, *Event*, *Barrier*, *Queue*, *Value* and *Array*. For example,

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))

        p = Process(target=f, args=(d, l))
        p.start()
        p.join()

        print(d)
        print(l)
```

será impresso:

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Server process managers are more flexible than using shared memory objects because they can be made to support arbitrary object types. Also, a single manager can be shared by processes on different computers over a network. They are, however, slower than using shared memory.

Using a pool of workers

The *Pool* class represents a pool of worker processes. It has methods which allows tasks to be offloaded to the worker processes in a few different ways.

Por exemplo:

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    # start 4 worker processes
    with Pool(processes=4) as pool:

        # print "[0, 1, 4,..., 81]"
        print(pool.map(f, range(10)))

        # print same numbers in arbitrary order
        for i in pool.imap_unordered(f, range(10)):
            print(i)
```

(continua na próxima página)

(continuação da página anterior)

```
        # evaluate "f(20)" asynchronously
        res = pool.apply_async(f, (20,))      # runs in *only* one process
        print(res.get(timeout=1))             # prints "400"

        # evaluate "os.getpid()" asynchronously
        res = pool.apply_async(os.getpid, ()) # runs in *only* one process
        print(res.get(timeout=1))             # prints the PID of that process

        # launching multiple evaluations asynchronously *may* use more processes
        multiple_results = [pool.apply_async(os.getpid, ()) for i in range(4)]
        print([res.get(timeout=1) for res in multiple_results])

        # make a single worker sleep for 10 secs
        res = pool.apply_async(time.sleep, (10,))
        try:
            print(res.get(timeout=1))
        except TimeoutError:
            print("We lacked patience and got a multiprocessing.TimeoutError")

        print("For the moment, the pool remains available for more work")

    # exiting the 'with'-block has stopped the pool
    print("Now the pool is closed and no longer available")
```

Note that the methods of a pool should only ever be used by the process which created it.

Nota: Functionality within this package requires that the `__main__` module be importable by the children. This is covered in *Programming guidelines* however it is worth pointing out here. This means that some examples, such as the *`multiprocessing.pool.Pool`* examples will not work in the interactive interpreter. For example:

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> with p:
...   p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
```

(If you try this it will actually output three full tracebacks interleaved in a semi-random fashion, and then you may have to stop the master process somehow.)

17.2.2 Referência

The *multiprocessing* package mostly replicates the API of the *threading* module.

Process and exceptions

class multiprocessing.**Process**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, ***, *daemon=None*)

Objetos de processo representam atividades que são executadas em um processo separado. A classe `processo` possui equivalentes de todos os métodos de *threading.Thread*.

The constructor should always be called with keyword arguments. *group* should always be `None`; it exists solely for compatibility with *threading.Thread*. *target* is the callable object to be invoked by the *run()* method. It defaults to `None`, meaning nothing is called. *name* is the process name (see *name* for more details). *args* is the argument tuple for the target invocation. *kwargs* is a dictionary of keyword arguments for the target invocation. If provided, the keyword-only *daemon* argument sets the process *daemon* flag to `True` or `False`. If `None` (the default), this flag will be inherited from the creating process.

Por padrão, nenhum argumento é passado para *target*.

If a subclass overrides the constructor, it must make sure it invokes the base class constructor (`Process.__init__()`) before doing anything else to the process.

Alterado na versão 3.3: Adicionado o argumento *daemon*.

run()

Método que representa a atividade do processo.

You may override this method in a subclass. The standard *run()* method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the *args* and *kwargs* arguments, respectively.

start()

Inicie a atividade do processo.

This must be called at most once per process object. It arranges for the object's *run()* method to be invoked in a separate process.

join([*timeout*])

If the optional argument *timeout* is `None` (the default), the method blocks until the process whose *join()* method is called terminates. If *timeout* is a positive number, it blocks at most *timeout* seconds. Note that the method returns `None` if its process terminates or if the method times out. Check the process's *exitcode* to determine if it terminated.

A process can be joined many times.

A process cannot join itself because this would cause a deadlock. It is an error to attempt to join a process before it has been started.

name

The process's name. The name is a string used for identification purposes only. It has no semantics. Multiple processes may be given the same name.

The initial name is set by the constructor. If no explicit name is provided to the constructor, a name of the form 'Process-N_1:N_2:…:N_k' is constructed, where each N_k is the N-th child of its parent.

is_alive()

Retorne se o processo está ativo.

Roughly, a process object is alive from the moment the *start()* method returns until the child process terminates.

daemon
The process's daemon flag, a Boolean value. This must be set before *start()* is called.

The initial value is inherited from the creating process.

When a process exits, it attempts to terminate all of its daemonic child processes.

Note that a daemonic process is not allowed to create child processes. Otherwise a daemonic process would leave its children orphaned if it gets terminated when its parent process exits. Additionally, these are **not** Unix daemons or services, they are normal processes that will be terminated (and not joined) if non-daemonic processes have exited.

In addition to the *threading.Thread* API, *Process* objects also support the following attributes and methods:

pid
Return the process ID. Before the process is spawned, this will be `None`.

exitcode
The child's exit code. This will be `None` if the process has not yet terminated. A negative value *-N* indicates that the child was terminated by signal *N*.

authkey
The process's authentication key (a byte string).

When *multiprocessing* is initialized the main process is assigned a random string using *os.urandom()*.

When a *Process* object is created, it will inherit the authentication key of its parent process, although this may be changed by setting *authkey* to another byte string.

See *Authentication keys*.

sentinel
A numeric handle of a system object which will become "ready" when the process ends.

You can use this value if you want to wait on several events at once using *multiprocessing.connection.wait()*. Otherwise calling *join()* is simpler.

On Windows, this is an OS handle usable with the `WaitForSingleObject` and `WaitForMultipleObjects` family of API calls. On Unix, this is a file descriptor usable with primitives from the *select* module.

Novo na versão 3.3.

terminate()
Terminate the process. On Unix this is done using the `SIGTERM` signal; on Windows `TerminateProcess()` is used. Note that exit handlers and finally clauses, etc., will not be executed.

Note that descendant processes of the process will *not* be terminated – they will simply become orphaned.

> **Aviso:** If this method is used when the associated process is using a pipe or queue then the pipe or queue is liable to become corrupted and may become unusable by other process. Similarly, if the process has acquired a lock or semaphore etc. then terminating it is liable to cause other processes to deadlock.

kill()
Same as *terminate()* but using the `SIGKILL` signal on Unix.

Novo na versão 3.7.

close()
Close the *Process* object, releasing all resources associated with it. *ValueError* is raised if the underlying process is still running. Once *close()* returns successfully, most other methods and attributes of the *Process* object will raise *ValueError*.

Novo na versão 3.7.

Note that the *start()*, *join()*, *is_alive()*, *terminate()* and *exitcode* methods should only be called by the process that created the process object.

Example usage of some of the methods of *Process*:

```
>>> import multiprocessing, time, signal
>>> p = multiprocessing.Process(target=time.sleep, args=(1000,))
>>> print(p, p.is_alive())
<Process(Process-1, initial)> False
>>> p.start()
>>> print(p, p.is_alive())
<Process(Process-1, started)> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<Process(Process-1, stopped[SIGTERM])> False
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.**ProcessError**
The base class of all *multiprocessing* exceptions.

exception multiprocessing.**BufferTooShort**
Exception raised by `Connection.recv_bytes_into()` when the supplied buffer object is too small for the message read.

If `e` is an instance of *BufferTooShort* then `e.args[0]` will give the message as a byte string.

exception multiprocessing.**AuthenticationError**
Raised when there is an authentication error.

exception multiprocessing.**TimeoutError**
Raised by methods with a timeout when the timeout expires.

Pipes and Queues

When using multiple processes, one generally uses message passing for communication between processes and avoids having to use any synchronization primitives like locks.

For passing messages one can use *Pipe()* (for a connection between two processes) or a queue (which allows multiple producers and consumers).

The *Queue*, *SimpleQueue* and *JoinableQueue* types are multi-producer, multi-consumer FIFO queues modelled on the *queue.Queue* class in the standard library. They differ in that *Queue* lacks the *task_done()* and *join()* methods introduced into Python 2.5's *queue.Queue* class.

If you use *JoinableQueue* then you **must** call *JoinableQueue.task_done()* for each task removed from the queue or else the semaphore used to count the number of unfinished tasks may eventually overflow, raising an exception.

Note that one can also create a shared queue by using a manager object – see *Gerenciadores*.

Nota: *multiprocessing* uses the usual *queue.Empty* and *queue.Full* exceptions to signal a timeout. They are not available in the *multiprocessing* namespace so you need to import them from *queue*.

Nota: When an object is put on a queue, the object is pickled and a background thread later flushes the pickled data to an underlying pipe. This has some consequences which are a little surprising, but should not cause any practical difficulties – if they really bother you then you can instead use a queue created with a *manager*.

(1) After putting an object on an empty queue there may be an infinitesimal delay before the queue's *empty()* method returns *False* and *get_nowait()* can return without raising *queue.Empty*.

(2) If multiple processes are enqueuing objects, it is possible for the objects to be received at the other end out-of-order. However, objects enqueued by the same process will always be in the expected order with respect to each other.

Aviso: If a process is killed using *Process.terminate()* or *os.kill()* while it is trying to use a *Queue*, then the data in the queue is likely to become corrupted. This may cause any other process to get an exception when it tries to use the queue later on.

Aviso: As mentioned above, if a child process has put items on a queue (and it has not used *JoinableQueue.cancel_join_thread*), then that process will not terminate until all buffered items have been flushed to the pipe.

This means that if you try joining that process you may get a deadlock unless you are sure that all items which have been put on the queue have been consumed. Similarly, if the child process is non-daemonic then the parent process may hang on exit when it tries to join all its non-daemonic children.

Note that a queue created using a manager does not have this issue. See *Programming guidelines*.

For an example of the usage of queues for interprocess communication see *Exemplos*.

`multiprocessing.`**`Pipe`**`(`[*duplex*]`)`

Returns a pair `(conn1, conn2)` of *Connection* objects representing the ends of a pipe.

If *duplex* is `True` (the default) then the pipe is bidirectional. If *duplex* is `False` then the pipe is unidirectional: `conn1` can only be used for receiving messages and `conn2` can only be used for sending messages.

`class` `multiprocessing.`**`Queue`**`(`[*maxsize*]`)`

Returns a process shared queue implemented using a pipe and a few locks/semaphores. When a process first puts an item on the queue a feeder thread is started which transfers objects from a buffer into the pipe.

The usual *queue.Empty* and *queue.Full* exceptions from the standard library's *queue* module are raised to signal timeouts.

Queue implements all the methods of *queue.Queue* except for *task_done()* and *join()*.

`qsize``()`

Return the approximate size of the queue. Because of multithreading/multiprocessing semantics, this number is not reliable.

Note that this may raise *NotImplementedError* on Unix platforms like Mac OS X where `sem_getvalue()` is not implemented.

`empty``()`

Return `True` if the queue is empty, `False` otherwise. Because of multithreading/multiprocessing semantics,

this is not reliable.

full()
Return `True` if the queue is full, `False` otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.

put(*obj*[, *block*[, *timeout*]])
Put obj into the queue. If the optional argument *block* is `True` (the default) and *timeout* is `None` (the default), block if necessary until a free slot is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *queue.Full* exception if no free slot was available within that time. Otherwise (*block* is `False`), put an item on the queue if a free slot is immediately available, else raise the *queue.Full* exception (*timeout* is ignored in that case).

put_nowait(*obj*)
Equivalent to `put(obj, False)`.

get([*block*[, *timeout*]])
Remove and return an item from the queue. If optional args *block* is `True` (the default) and *timeout* is `None` (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *queue.Empty* exception if no item was available within that time. Otherwise (block is `False`), return an item if one is immediately available, else raise the *queue.Empty* exception (*timeout* is ignored in that case).

get_nowait()
Equivalente a `get(False)`.

multiprocessing.Queue has a few additional methods not found in *queue.Queue*. These methods are usually unnecessary for most code:

close()
Indicate that no more data will be put on this queue by the current process. The background thread will quit once it has flushed all buffered data to the pipe. This is called automatically when the queue is garbage collected.

join_thread()
Join the background thread. This can only be used after *close()* has been called. It blocks until the background thread exits, ensuring that all data in the buffer has been flushed to the pipe.

By default if a process is not the creator of the queue then on exit it will attempt to join the queue's background thread. The process can call *cancel_join_thread()* to make *join_thread()* do nothing.

cancel_join_thread()
Prevent *join_thread()* from blocking. In particular, this prevents the background thread from being joined automatically when the process exits – see *join_thread()*.

A better name for this method might be `allow_exit_without_flush()`. It is likely to cause enqueued data to lost, and you almost certainly will not need to use it. It is really only there if you need the current process to exit immediately without waiting to flush enqueued data to the underlying pipe, and you don't care about lost data.

Nota: This class's functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the functionality in this class will be disabled, and attempts to instantiate a *Queue* will result in an *ImportError*. See bpo-3770 for additional information. The same holds true for any of the specialized queue types listed below.

---

class multiprocessing.**SimpleQueue**
It is a simplified *Queue* type, very close to a locked *Pipe*.

empty()
Retorna `True` se a fila estiver vazia, `False` caso contrário.

get()
Remove and return an item from the queue.

put(*item*)
Put *item* into the queue.

class multiprocessing.**JoinableQueue**([*maxsize*])
JoinableQueue, a *Queue* subclass, is a queue which additionally has *task_done()* and *join()* methods.

task_done()
Indicate that a formerly enqueued task is complete. Used by queue consumers. For each *get()* used to fetch a task, a subsequent call to *task_done()* tells the queue that the processing on the task is complete.

If a *join()* is currently blocking, it will resume when all items have been processed (meaning that a *task_done()* call was received for every item that had been *put()* into the queue).

Raises a *ValueError* if called more times than there were items placed in the queue.

join()
Block until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer calls *task_done()* to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, *join()* unblocks.

Diversos

multiprocessing.**active_children**()
Return list of all live children of the current process.

Calling this has the side effect of "joining" any processes which have already finished.

multiprocessing.**cpu_count**()
Return the number of CPUs in the system.

Este número não é equivalente ao número de CPUs que o processo atual pode usar. O número de CPUs utilizáveis pode ser obtido com `len(os.sched_getaffinity(0))`

May raise *NotImplementedError*.

Ver também:

os.cpu_count()

multiprocessing.**current_process**()
Return the *Process* object corresponding to the current process.

An analogue of *threading.current_thread()*.

multiprocessing.**freeze_support**()
Add support for when a program which uses *multiprocessing* has been frozen to produce a Windows executable. (Has been tested with **py2exe**, **PyInstaller** and **cx_Freeze**.)

One needs to call this function straight after the `if __name__ == '__main__'` line of the main module. For example:

```
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

If the `freeze_support()` line is omitted then trying to run the frozen executable will raise *RuntimeError*.

Calling `freeze_support()` has no effect when invoked on any operating system other than Windows. In addition, if the module is being run normally by the Python interpreter on Windows (the program has not been frozen), then `freeze_support()` has no effect.

`multiprocessing.`**`get_all_start_methods`**`()`
Returns a list of the supported start methods, the first of which is the default. The possible start methods are `'fork'`, `'spawn'` and `'forkserver'`. On Windows only `'spawn'` is available. On Unix `'fork'` and `'spawn'` are always supported, with `'fork'` being the default.

Novo na versão 3.4.

`multiprocessing.`**`get_context`**`(`*method=None*`)`
Return a context object which has the same attributes as the *multiprocessing* module.

If *method* is `None` then the default context is returned. Otherwise *method* should be `'fork'`, `'spawn'`, `'forkserver'`. *ValueError* is raised if the specified start method is not available.

Novo na versão 3.4.

`multiprocessing.`**`get_start_method`**`(`*allow_none=False*`)`
Return the name of start method used for starting processes.

If the start method has not been fixed and *allow_none* is false, then the start method is fixed to the default and the name is returned. If the start method has not been fixed and *allow_none* is true then `None` is returned.

The return value can be `'fork'`, `'spawn'`, `'forkserver'` or `None`. `'fork'` is the default on Unix, while `'spawn'` is the default on Windows.

Novo na versão 3.4.

`multiprocessing.`**`set_executable`**`()`
Sets the path of the Python interpreter to use when starting a child process. (By default *sys.executable* is used). Embedders will probably need to do some thing like

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

before they can create child processes.

Alterado na versão 3.4: Now supported on Unix when the `'spawn'` start method is used.

`multiprocessing.`**`set_start_method`**`(`*method*`)`
Set the method which should be used to start child processes. *method* can be `'fork'`, `'spawn'` or `'forkserver'`.

Note that this should be called at most once, and it should be protected inside the `if __name__ == '__main__'` clause of the main module.

Novo na versão 3.4.

Nota: *multiprocessing* contains no analogues of *threading.active_count()*, *threading.enumerate()*, *threading.settrace()*, *threading.setprofile()*, *threading.Timer*, or *threading.local*.

Connection Objects

Connection objects allow the sending and receiving of picklable objects or strings. They can be thought of as message oriented connected sockets.

Connection objects are usually created using *Pipe* – see also *Listeners and Clients*.

class multiprocessing.connection.**Connection**

send(*obj*)
Send an object to the other end of the connection which should be read using *recv()*.

The object must be picklable. Very large pickles (approximately 32 MiB+, though it depends on the OS) may raise a *ValueError* exception.

recv()
Return an object sent from the other end of the connection using *send()*. Blocks until there is something to receive. Raises *EOFError* if there is nothing left to receive and the other end was closed.

fileno()
Return the file descriptor or handle used by the connection.

close()
Close the connection.

This is called automatically when the connection is garbage collected.

poll([*timeout*])
Return whether there is any data available to be read.

If *timeout* is not specified then it will return immediately. If *timeout* is a number then this specifies the maximum time in seconds to block. If *timeout* is None then an infinite timeout is used.

Note that multiple connection objects may be polled at once by using *multiprocessing.connection.wait()*.

send_bytes(*buffer*[, *offset*[, *size*]])
Send byte data from a *bytes-like object* as a complete message.

If *offset* is given then data is read from that position in *buffer*. If *size* is given then that many bytes will be read from buffer. Very large buffers (approximately 32 MiB+, though it depends on the OS) may raise a *ValueError* exception

recv_bytes([*maxlength*])
Return a complete message of byte data sent from the other end of the connection as a string. Blocks until there is something to receive. Raises *EOFError* if there is nothing left to receive and the other end has closed.

If *maxlength* is specified and the message is longer than *maxlength* then *OSError* is raised and the connection will no longer be readable.

Alterado na versão 3.3: This function used to raise *IOError*, which is now an alias of *OSError*.

recv_bytes_into(*buffer*[, *offset*])
Read into *buffer* a complete message of byte data sent from the other end of the connection and return the number of bytes in the message. Blocks until there is something to receive. Raises *EOFError* if there is nothing left to receive and the other end was closed.

buffer must be a writable *bytes-like object*. If *offset* is given then the message will be written into the buffer from that position. Offset must be a non-negative integer less than the length of *buffer* (in bytes).

If the buffer is too short then a `BufferTooShort` exception is raised and the complete message is available as `e.args[0]` where `e` is the exception instance.

Alterado na versão 3.3: Connection objects themselves can now be transferred between processes using *Connection.send()* and *Connection.recv()*.

Novo na versão 3.3: Connection objects now support the context management protocol – see *Tipos de Gerenciador de Contexto*. *__enter__()* returns the connection object, and *__exit__()* calls *close()*.

Por exemplo:

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
b'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

Aviso: The *Connection.recv()* method automatically unpickles the data it receives, which can be a security risk unless you can trust the process which sent the message.

Therefore, unless the connection object was produced using `Pipe()` you should only use the *recv()* and *send()* methods after performing some sort of authentication. See *Authentication keys*.

Aviso: If a process is killed while it is trying to read or write to a pipe then the data in the pipe is likely to become corrupted, because it may become impossible to be sure where the message boundaries lie.

Synchronization primitives

Generally synchronization primitives are not as necessary in a multiprocess program as they are in a multithreaded program. See the documentation for *threading* module.

Note that one can also create synchronization primitives by using a manager object – see *Gerenciadores*.

class multiprocessing.**Barrier**(*parties*[, *action*[, *timeout*]])
A barrier object: a clone of *threading.Barrier*.

Novo na versão 3.3.

class multiprocessing.**BoundedSemaphore**([*value*])
A bounded semaphore object: a close analog of *threading.BoundedSemaphore*.

A solitary difference from its close analog exists: its `acquire` method's first argument is named *block*, as is consistent with *Lock.acquire()*.

Nota: On Mac OS X, this is indistinguishable from *Semaphore* because `sem_getvalue()` is not implemented on that platform.

---

class multiprocessing.**Condition**([*lock*])
A condition variable: an alias for *threading.Condition*.

If *lock* is specified then it should be a *Lock* or *RLock* object from *multiprocessing*.

Alterado na versão 3.3: The *wait_for()* method was added.

class multiprocessing.**Event**
A clone of *threading.Event*.

class multiprocessing.**Lock**
A non-recursive lock object: a close analog of *threading.Lock*. Once a process or thread has acquired a lock, subsequent attempts to acquire it from any process or thread will block until it is released; any process or thread may release it. The concepts and behaviors of *threading.Lock* as it applies to threads are replicated here in *multiprocessing.Lock* as it applies to either processes or threads, except as noted.

Note that *Lock* is actually a factory function which returns an instance of `multiprocessing.synchronize.Lock` initialized with a default context.

Lock supports the *context manager* protocol and thus may be used in `with` statements.

acquire(*block=True*, *timeout=None*)
Acquire a lock, blocking or non-blocking.

With the *block* argument set to `True` (the default), the method call will block until the lock is in an unlocked state, then set it to locked and return `True`. Note that the name of this first argument differs from that in *threading.Lock.acquire()*.

With the *block* argument set to `False`, the method call does not block. If the lock is currently in a locked state, return `False`; otherwise set the lock to a locked state and return `True`.

When invoked with a positive, floating-point value for *timeout*, block for at most the number of seconds specified by *timeout* as long as the lock can not be acquired. Invocations with a negative value for *timeout* are equivalent to a *timeout* of zero. Invocations with a *timeout* value of `None` (the default) set the timeout period to infinite. Note that the treatment of negative or `None` values for *timeout* differs from the implemented behavior in *threading.Lock.acquire()*. The *timeout* argument has no practical implications if the *block* argument is set to `False` and is thus ignored. Returns `True` if the lock has been acquired or `False` if the timeout period has elapsed.

release()
Release a lock. This can be called from any process or thread, not only the process or thread which originally acquired the lock.

Behavior is the same as in *threading.Lock.release()* except that when invoked on an unlocked lock, a *ValueError* is raised.

class multiprocessing.**RLock**
A recursive lock object: a close analog of *threading.RLock*. A recursive lock must be released by the process or thread that acquired it. Once a process or thread has acquired a recursive lock, the same process or thread may acquire it again without blocking; that process or thread must release it once for each time it has been acquired.

Note that *RLock* is actually a factory function which returns an instance of multiprocessing.synchronize.RLock initialized with a default context.

RLock supports the *context manager* protocol and thus may be used in with statements.

acquire(*block=True*, *timeout=None*)
Acquire a lock, blocking or non-blocking.

When invoked with the *block* argument set to True, block until the lock is in an unlocked state (not owned by any process or thread) unless the lock is already owned by the current process or thread. The current process or thread then takes ownership of the lock (if it does not already have ownership) and the recursion level inside the lock increments by one, resulting in a return value of True. Note that there are several differences in this first argument's behavior compared to the implementation of *threading.RLock.acquire()*, starting with the name of the argument itself.

When invoked with the *block* argument set to False, do not block. If the lock has already been acquired (and thus is owned) by another process or thread, the current process or thread does not take ownership and the recursion level within the lock is not changed, resulting in a return value of False. If the lock is in an unlocked state, the current process or thread takes ownership and the recursion level is incremented, resulting in a return value of True.

Use and behaviors of the *timeout* argument are the same as in *Lock.acquire()*. Note that some of these behaviors of *timeout* differ from the implemented behaviors in *threading.RLock.acquire()*.

release()
Release a lock, decrementing the recursion level. If after the decrement the recursion level is zero, reset the lock to unlocked (not owned by any process or thread) and if any other processes or threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling process or thread.

Only call this method when the calling process or thread owns the lock. An *AssertionError* is raised if this method is called by a process or thread other than the owner or if the lock is in an unlocked (unowned) state. Note that the type of exception raised in this situation differs from the implemented behavior in *threading.RLock.release()*.

class multiprocessing.**Semaphore**([*value*])
A semaphore object: a close analog of *threading.Semaphore*.

A solitary difference from its close analog exists: its acquire method's first argument is named *block*, as is consistent with *Lock.acquire()*.

Nota: On Mac OS X, sem_timedwait is unsupported, so calling acquire() with a timeout will emulate that function's behavior using a sleeping loop.

Nota: If the SIGINT signal generated by Ctrl-C arrives while the main thread is blocked by a call to BoundedSemaphore.acquire(), *Lock.acquire()*, *RLock.acquire()*, Semaphore.acquire(),

`Condition.acquire()` or `Condition.wait()` then the call will be immediately interrupted and *`KeyboardInterrupt`* will be raised.

This differs from the behaviour of *`threading`* where SIGINT will be ignored while the equivalent blocking calls are in progress.

Nota: Some of this package's functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the `multiprocessing.synchronize` module will be disabled, and attempts to import it will result in an *`ImportError`*. See bpo-3770 for additional information.

Shared `ctypes` Objects

It is possible to create shared objects using shared memory which can be inherited by child processes.

`multiprocessing.`**`Value`**(*typecode_or_type*, **args*, *lock=True*)

Return a *`ctypes`* object allocated from shared memory. By default the return value is actually a synchronized wrapper for the object. The object itself can be accessed via the *value* attribute of a *`Value`*.

typecode_or_type determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the *`array`* module. **args* is passed on to the constructor for the type.

If *lock* is `True` (the default) then a new recursive lock object is created to synchronize access to the value. If *lock* is a *`Lock`* or *`RLock`* object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be "process-safe".

Operations like += which involve a read and write are not atomic. So if, for instance, you want to atomically increment a shared value it is insufficient to just do

```
counter.value += 1
```

Assuming the associated lock is recursive (which it is by default) you can instead do

```
with counter.get_lock():
    counter.value += 1
```

Note that *lock* is a keyword-only argument.

`multiprocessing.`**`Array`**(*typecode_or_type*, *size_or_initializer*, ***, *lock=True*)

Return a ctypes array allocated from shared memory. By default the return value is actually a synchronized wrapper for the array.

typecode_or_type determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the *`array`* module. If *size_or_initializer* is an integer, then it determines the length of the array, and the array will be initially zeroed. Otherwise, *size_or_initializer* is a sequence which is used to initialize the array and whose length determines the length of the array.

If *lock* is `True` (the default) then a new lock object is created to synchronize access to the value. If *lock* is a *`Lock`* or *`RLock`* object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be "process-safe".

Note that *lock* is a keyword only argument.

Note that an array of *`ctypes.c_char`* has *value* and *raw* attributes which allow one to use it to store and retrieve strings.

The `multiprocessing.sharedctypes` module

The *multiprocessing.sharedctypes* module provides functions for allocating *ctypes* objects from shared memory which can be inherited by child processes.

Nota: Although it is possible to store a pointer in shared memory remember that this will refer to a location in the address space of a specific process. However, the pointer is quite likely to be invalid in the context of a second process and trying to dereference the pointer from the second process may cause a crash.

---

multiprocessing.sharedctypes.**RawArray**(*typecode_or_type*, *size_or_initializer*)
Return a ctypes array allocated from shared memory.

typecode_or_type determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the *array* module. If *size_or_initializer* is an integer then it determines the length of the array, and the array will be initially zeroed. Otherwise *size_or_initializer* is a sequence which is used to initialize the array and whose length determines the length of the array.

Note that setting and getting an element is potentially non-atomic – use *Array()* instead to make sure that access is automatically synchronized using a lock.

multiprocessing.sharedctypes.**RawValue**(*typecode_or_type*, **args*)
Return a ctypes object allocated from shared memory.

typecode_or_type determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the *array* module. **args* is passed on to the constructor for the type.

Note that setting and getting the value is potentially non-atomic – use *Value()* instead to make sure that access is automatically synchronized using a lock.

Note that an array of *ctypes.c_char* has `value` and `raw` attributes which allow one to use it to store and retrieve strings – see documentation for *ctypes*.

multiprocessing.sharedctypes.**Array**(*typecode_or_type*, *size_or_initializer*, *, *lock=True*)
The same as *RawArray()* except that depending on the value of *lock* a process-safe synchronization wrapper may be returned instead of a raw ctypes array.

If *lock* is `True` (the default) then a new lock object is created to synchronize access to the value. If *lock* is a *Lock* or *RLock* object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be "process-safe".

Note that *lock* is a keyword-only argument.

multiprocessing.sharedctypes.**Value**(*typecode_or_type*, **args*, *lock=True*)
The same as *RawValue()* except that depending on the value of *lock* a process-safe synchronization wrapper may be returned instead of a raw ctypes object.

If *lock* is `True` (the default) then a new lock object is created to synchronize access to the value. If *lock* is a *Lock* or *RLock* object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be "process-safe".

Note that *lock* is a keyword-only argument.

multiprocessing.sharedctypes.**copy**(*obj*)
Return a ctypes object allocated from shared memory which is a copy of the ctypes object *obj*.

multiprocessing.sharedctypes.**synchronized**(*obj*[, *lock*])
Return a process-safe wrapper object for a ctypes object which uses *lock* to synchronize access. If *lock* is `None` (the default) then a *multiprocessing.RLock* object is created automatically.

A synchronized wrapper will have two methods in addition to those of the object it wraps: `get_obj()` returns the wrapped object and `get_lock()` returns the lock object used for synchronization.

Note that accessing the ctypes object through the wrapper can be a lot slower than accessing the raw ctypes object.

Alterado na versão 3.5: Synchronized objects support the *context manager* protocol.

The table below compares the syntax for creating shared ctypes objects from shared memory with the normal ctypes syntax. (In the table `MyStruct` is some subclass of *ctypes.Structure*.)

ctypes	sharedctypes using type	sharedctypes using typecode
c_double(2.4)	RawValue(c_double, 2.4)	RawValue('d', 2.4)
MyStruct(4, 6)	RawValue(MyStruct, 4, 6)	
(c_short * 7)()	RawArray(c_short, 7)	RawArray('h', 7)
(c_int * 3)(9, 2, 8)	RawArray(c_int, (9, 2, 8))	RawArray('i', (9, 2, 8))

Below is an example where a number of ctypes objects are modified by a child process:

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', b'hello world', lock=lock)
    A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()

    print(n.value)
    print(x.value)
    print(s.value)
    print([(a.x, a.y) for a in A])
```

The results printed are

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

Gerenciadores

Managers provide a way to create data which can be shared between different processes, including sharing over a network between processes running on different machines. A manager object controls a server process which manages *shared objects*. Other processes can access the shared objects by using proxies.

`multiprocessing.`**`Manager`**`()`
Returns a started *SyncManager* object which can be used for sharing objects between processes. The returned manager object corresponds to a spawned child process and has methods which will create shared objects and return corresponding proxies.

Manager processes will be shutdown as soon as they are garbage collected or their parent process exits. The manager classes are defined in the *multiprocessing.managers* module:

class `multiprocessing.managers.`**`BaseManager`**`(`[*address*[, *authkey*]]`)`
Criando um objeto BaseManager.

Once created one should call *start()* or `get_server().serve_forever()` to ensure that the manager object refers to a started manager process.

address is the address on which the manager process listens for new connections. If *address* is `None` then an arbitrary one is chosen.

authkey is the authentication key which will be used to check the validity of incoming connections to the server process. If *authkey* is `None` then `current_process().authkey` is used. Otherwise *authkey* is used and it must be a byte string.

`start``(`[*initializer*[, *initargs*]]`)`
Start a subprocess to start the manager. If *initializer* is not `None` then the subprocess will call `initializer(*initargs)` when it starts.

`get_server``()`
Returns a `Server` object which represents the actual server under the control of the Manager. The `Server` object supports the `serve_forever()` method:

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000), authkey=b'abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

`Server` additionally has an *address* attribute.

`connect``()`
Connect a local manager object to a remote manager process:

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 50000), authkey=b'abc')
>>> m.connect()
```

`shutdown``()`
Stop the process used by the manager. This is only available if *start()* has been used to start the server process.

This can be called multiple times.

`register``(`*typeid*[, *callable*[, *proxytype*[, *exposed*[, *method_to_typeid*[, *create_method*]]]]]`)`
A classmethod which can be used for registering a type or callable with the manager class.

typeid is a "type identifier" which is used to identify a particular type of shared object. This must be a string.

callable is a callable used for creating objects for this type identifier. If a manager instance will be connected to the server using the `connect()` method, or if the *create_method* argument is `False` then this can be left as `None`.

proxytype is a subclass of `BaseProxy` which is used to create proxies for shared objects with this *typeid*. If `None` then a proxy class is created automatically.

exposed is used to specify a sequence of method names which proxies for this typeid should be allowed to access using `BaseProxy._callmethod()`. (If *exposed* is `None` then `proxytype._exposed_` is used instead if it exists.) In the case where no exposed list is specified, all "public methods" of the shared object will be accessible. (Here a "public method" means any attribute which has a `__call__()` method and whose name does not begin with `'_'`.)

method_to_typeid is a mapping used to specify the return type of those exposed methods which should return a proxy. It maps method names to typeid strings. (If *method_to_typeid* is `None` then `proxytype._method_to_typeid_` is used instead if it exists.) If a method's name is not a key of this mapping or if the mapping is `None` then the object returned by the method will be copied by value.

create_method determines whether a method should be created with name *typeid* which can be used to tell the server process to create a new shared object and return a proxy for it. By default it is `True`.

`BaseManager` instances also have one read-only property:

address
The address used by the manager.

Alterado na versão 3.3: Manager objects support the context management protocol – see *Tipos de Gerenciador de Contexto*. `__enter__()` starts the server process (if it has not already started) and then returns the manager object. `__exit__()` calls `shutdown()`.

In previous versions `__enter__()` did not start the manager's server process if it was not already started.

class multiprocessing.managers.**SyncManager**
A subclass of `BaseManager` which can be used for the synchronization of processes. Objects of this type are returned by `multiprocessing.Manager()`.

Its methods create and return *Proxy Objects* for a number of commonly used data types to be synchronized across processes. This notably includes shared lists and dictionaries.

Barrier(*parties*[, *action*[, *timeout*]])
Create a shared `threading.Barrier` object and return a proxy for it.

Novo na versão 3.3.

BoundedSemaphore([*value*])
Create a shared `threading.BoundedSemaphore` object and return a proxy for it.

Condition([*lock*])
Create a shared `threading.Condition` object and return a proxy for it.

If *lock* is supplied then it should be a proxy for a `threading.Lock` or `threading.RLock` object.

Alterado na versão 3.3: The `wait_for()` method was added.

Event()
Create a shared `threading.Event` object and return a proxy for it.

Lock()
Create a shared `threading.Lock` object and return a proxy for it.

Namespace()
Create a shared `Namespace` object and return a proxy for it.

Queue([*maxsize*])
Create a shared *queue.Queue* object and return a proxy for it.

RLock()
Create a shared *threading.RLock* object and return a proxy for it.

Semaphore([*value*])
Create a shared *threading.Semaphore* object and return a proxy for it.

Array(*typecode*, *sequence*)
Create an array and return a proxy for it.

Value(*typecode*, *value*)
Create an object with a writable `value` attribute and return a proxy for it.

dict()
dict(*mapping*)
dict(*sequence*)
Create a shared *dict* object and return a proxy for it.

list()
list(*sequence*)
Create a shared *list* object and return a proxy for it.

Alterado na versão 3.6: Shared objects are capable of being nested. For example, a shared container object such as a shared list can contain other shared objects which will all be managed and synchronized by the *SyncManager*.

class multiprocessing.managers.**Namespace**
A type that can register with *SyncManager*.

A namespace object has no public methods, but does have writable attributes. Its representation shows the values of its attributes.

However, when using a proxy for a namespace object, an attribute beginning with `'_'` will be an attribute of the proxy and not an attribute of the referent:

```
>>> manager = multiprocessing.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3    # this is an attribute of the proxy
>>> print(Global)
Namespace(x=10, y='hello')
```

Customized managers

To create one's own manager, one creates a subclass of *BaseManager* and uses the *register()* classmethod to register new types or callables with the manager class. For example:

```
from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
```

(continua na próxima página)

(continuação da página anterior)

```
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))         # prints 7
        print(maths.mul(7, 8))         # prints 56
```

Using a remote manager

It is possible to run a manager server on one machine and have clients use it from other machines (assuming that the firewalls involved allow it).

Running the following commands creates a server for a single shared queue which remote clients can access:

```
>>> from multiprocessing.managers import BaseManager
>>> from queue import Queue
>>> queue = Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

One client can access the server as follows:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')
```

Another client can also use it:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

Local processes can also access that queue, using the code from above on the client to access it remotely:

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super(Worker, self).__init__()
```

(continua na próxima página)

(continuação da página anterior)

```
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

Proxy Objects

A proxy is an object which *refers* to a shared object which lives (presumably) in a different process. The shared object is said to be the *referent* of the proxy. Multiple proxy objects may have the same referent.

A proxy object has methods which invoke corresponding methods of its referent (although not every method of the referent will necessarily be available through the proxy). In this way, a proxy can be used just like its referent can:

```
>>> from multiprocessing import Manager
>>> manager = Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print(l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print(repr(l))
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

Notice that applying *str()* to a proxy will return the representation of the referent, whereas applying *repr()* will return the representation of the proxy.

An important feature of proxy objects is that they are picklable so they can be passed between processes. As such, a referent can contain *Proxy Objects*. This permits nesting of these managed lists, dicts, and other *Proxy Objects*:

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)         # referent of a now contains referent of b
>>> print(a, b)
[<ListProxy object, typeid 'list' at ...>] []
>>> b.append('hello')
>>> print(a[0], b)
['hello'] ['hello']
```

Similarly, dict and list proxies may be nested inside one another:

```
>>> l_outer = manager.list([ manager.dict() for i in range(2) ])
>>> d_first_inner = l_outer[0]
>>> d_first_inner['a'] = 1
>>> d_first_inner['b'] = 2
>>> l_outer[1]['c'] = 3
```

(continua na próxima página)

(continuação da página anterior)

```
>>> l_outer[1]['z'] = 26
>>> print(l_outer[0])
{'a': 1, 'b': 2}
>>> print(l_outer[1])
{'c': 3, 'z': 26}
```

If standard (non-proxy) *list* or *dict* objects are contained in a referent, modifications to those mutable values will not be propagated through the manager because the proxy has no way of knowing when the values contained within are modified. However, storing a value in a container proxy (which triggers a `__setitem__` on the proxy object) does propagate through the manager and so to effectively modify such an item, one could re-assign the modified value to the container proxy:

```
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# updating the dictionary, the proxy is notified of the change
lproxy[0] = d
```

This approach is perhaps less convenient than employing nested *Proxy Objects* for most use cases but also demonstrates a level of control over the synchronization.

Nota: The proxy types in *multiprocessing* do nothing to support comparisons by value. So, for instance, we have:

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

One should just use a copy of the referent instead when making comparisons.

---

class multiprocessing.managers.**BaseProxy**

Proxy objects are instances of subclasses of *BaseProxy*.

_callmethod(*methodname*[, *args*[, *kwds*]])

Call and return the result of a method of the proxy's referent.

If `proxy` is a proxy whose referent is `obj` then the expression

```
proxy._callmethod(methodname, args, kwds)
```

will evaluate the expression

```
getattr(obj, methodname)(*args, **kwds)
```

in the manager's process.

The returned value will be a copy of the result of the call or a proxy to a new shared object – see documentation for the *method_to_typeid* argument of *BaseManager.register()*.

If an exception is raised by the call, then is re-raised by *_callmethod()*. If some other exception is raised in the manager's process then this is converted into a `RemoteError` exception and is raised by *_callmethod()*.

Note in particular that an exception will be raised if *methodname* has not been *exposed*.

An example of the usage of *_callmethod()*:

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getitem__', (slice(2, 7),)) # equivalent to l[2:7]
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))          # equivalent to l[20]
Traceback (most recent call last):
...
IndexError: list index out of range
```

_getvalue()
Return a copy of the referent.

If the referent is unpicklable then this will raise an exception.

__repr__()
Return a representation of the proxy object.

__str__()
Return the representation of the referent.

Cleanup

A proxy object uses a weakref callback so that when it gets garbage collected it deregisters itself from the manager which owns its referent.

A shared object gets deleted from the manager process when there are no longer any proxies referring to it.

Process Pools

One can create a pool of processes which will carry out tasks submitted to it with the *Pool* class.

class multiprocessing.pool.**Pool**([*processes*[, *initializer*[, *initargs*[, *maxtasksperchild*[, *context*]]]]])

A process pool object which controls a pool of worker processes to which jobs can be submitted. It supports asynchronous results with timeouts and callbacks and has a parallel map implementation.

processes is the number of worker processes to use. If *processes* is `None` then the number returned by *os.cpu_count()* is used.

If *initializer* is not `None` then each worker process will call `initializer(*initargs)` when it starts.

maxtasksperchild is the number of tasks a worker process can complete before it will exit and be replaced with a fresh worker process, to enable unused resources to be freed. The default *maxtasksperchild* is `None`, which means worker processes will live as long as the pool.

context can be used to specify the context used for starting the worker processes. Usually a pool is created using the function `multiprocessing.Pool()` or the *Pool()* method of a context object. In both cases *context* is set appropriately.

Note that the methods of the pool object should only be called by the process which created the pool.

Aviso: *multiprocessing.pool* objects have internal resources that need to be properly managed (like any other resource) by using the pool as a context manager or by calling *close()* and *terminate()* manually. Failure to do this can lead to the process hanging on finalization.

Note that is **not correct** to rely on the garbage colletor to destroy the pool as CPython does not assure that the finalizer of the pool will be called (see `object.__del__()` for more information).

Novo na versão 3.2: *maxtasksperchild*

Novo na versão 3.4: *context*

Nota: Worker processes within a `Pool` typically live for the complete duration of the Pool's work queue. A frequent pattern found in other systems (such as Apache, mod_wsgi, etc) to free resources held by workers is to allow a worker within a pool to complete only a set amount of work before being exiting, being cleaned up and a new process spawned to replace the old one. The *maxtasksperchild* argument to the `Pool` exposes this ability to the end user.

---

apply(*func*[, *args*[, *kwds*]])
Call *func* with arguments *args* and keyword arguments *kwds*. It blocks until the result is ready. Given this blocks, `apply_async()` is better suited for performing work in parallel. Additionally, *func* is only executed in one of the workers of the pool.

apply_async(*func*[, *args*[, *kwds*[, *callback*[, *error_callback*]]]])
A variant of the `apply()` method which returns a result object.

If *callback* is specified then it should be a callable which accepts a single argument. When the result becomes ready *callback* is applied to it, that is unless the call failed, in which case the *error_callback* is applied instead.

If *error_callback* is specified then it should be a callable which accepts a single argument. If the target function fails, then the *error_callback* is called with the exception instance.

Callbacks should complete immediately since otherwise the thread which handles the results will get blocked.

map(*func*, *iterable*[, *chunksize*])
A parallel equivalent of the `map()` built-in function (it supports only one *iterable* argument though, for multiple iterables see `starmap()`). It blocks until the result is ready.

This method chops the iterable into a number of chunks which it submits to the process pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer.

Note that it may cause high memory usage for very long iterables. Consider using `imap()` or `imap_unordered()` with explicit *chunksize* option for better efficiency.

map_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])
A variant of the `map()` method which returns a result object.

If *callback* is specified then it should be a callable which accepts a single argument. When the result becomes ready *callback* is applied to it, that is unless the call failed, in which case the *error_callback* is applied instead.

If *error_callback* is specified then it should be a callable which accepts a single argument. If the target function fails, then the *error_callback* is called with the exception instance.

Callbacks should complete immediately since otherwise the thread which handles the results will get blocked.

imap(*func*, *iterable*[, *chunksize*])
A lazier version of `map()`.

The *chunksize* argument is the same as the one used by the `map()` method. For very long iterables using a large value for *chunksize* can make the job complete **much** faster than using the default value of `1`.

Also if *chunksize* is `1` then the `next()` method of the iterator returned by the `imap()` method has an optional *timeout* parameter: `next(timeout)` will raise `multiprocessing.TimeoutError` if the result cannot be returned within *timeout* seconds.

imap_unordered(*func*, *iterable*[, *chunksize*])
The same as `imap()` except that the ordering of the results from the returned iterator should be considered arbitrary. (Only when there is only one worker process is the order guaranteed to be "correct".)

starmap(*func*, *iterable*[, *chunksize*])
Like `map()` except that the elements of the *iterable* are expected to be iterables that are unpacked as arguments.

Hence an *iterable* of `[(1,2), (3, 4)]` results in `[func(1,2), func(3,4)]`.

Novo na versão 3.3.

starmap_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])
A combination of `starmap()` and `map_async()` that iterates over *iterable* of iterables and calls *func* with the iterables unpacked. Returns a result object.

Novo na versão 3.3.

close()
Prevents any more tasks from being submitted to the pool. Once all the tasks have been completed the worker processes will exit.

terminate()
Stops the worker processes immediately without completing outstanding work. When the pool object is garbage collected `terminate()` will be called immediately.

join()
Wait for the worker processes to exit. One must call `close()` or `terminate()` before using `join()`.

Novo na versão 3.3: Pool objects now support the context management protocol – see *Tipos de Gerenciador de Contexto*. `__enter__()` returns the pool object, and `__exit__()` calls `terminate()`.

class multiprocessing.pool.**AsyncResult**
The class of the result returned by `Pool.apply_async()` and `Pool.map_async()`.

get([*timeout*])
Return the result when it arrives. If *timeout* is not `None` and the result does not arrive within *timeout* seconds then `multiprocessing.TimeoutError` is raised. If the remote call raised an exception then that exception will be reraised by `get()`.

wait([*timeout*])
Wait until the result is available or until *timeout* seconds pass.

ready()
Return whether the call has completed.

successful()
Return whether the call completed without raising an exception. Will raise `ValueError` if the result is not ready.

The following example demonstrates the use of a pool:

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:          # start 4 worker processes
        result = pool.apply_async(f, (10,)) # evaluate "f(10)" asynchronously in a␣
↪single process
```

(continua na próxima página)

(continuação da página anterior)

```
        print(result.get(timeout=1))         # prints "100" unless your computer is␣
↪*very* slow

        print(pool.map(f, range(10)))        # prints "[0, 1, 4,..., 81]"

        it = pool.imap(f, range(10))
        print(next(it))                      # prints "0"
        print(next(it))                      # prints "1"
        print(it.next(timeout=1))            # prints "4" unless your computer is␣
↪*very* slow

        result = pool.apply_async(time.sleep, (10,))
        print(result.get(timeout=1))         # raises multiprocessing.TimeoutError
```

Listeners and Clients

Usually message passing between processes is done using queues or by using *Connection* objects returned by *Pipe()*.

However, the *multiprocessing.connection* module allows some extra flexibility. It basically gives a high level message oriented API for dealing with sockets or Windows named pipes. It also has support for *digest authentication* using the *hmac* module, and for polling multiple connections at the same time.

multiprocessing.connection.**deliver_challenge**(*connection*, *authkey*)
Send a randomly generated message to the other end of the connection and wait for a reply.

If the reply matches the digest of the message using *authkey* as the key then a welcome message is sent to the other end of the connection. Otherwise *AuthenticationError* is raised.

multiprocessing.connection.**answer_challenge**(*connection*, *authkey*)
Receive a message, calculate the digest of the message using *authkey* as the key, and then send the digest back.

If a welcome message is not received, then *AuthenticationError* is raised.

multiprocessing.connection.**Client**(*address*[, *family*[, *authkey*]])
Attempt to set up a connection to the listener which is using address *address*, returning a *Connection*.

The type of the connection is determined by *family* argument, but this can generally be omitted since it can usually be inferred from the format of *address*. (See *Formatos de Endereços*)

If *authkey* is given and not None, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if *authkey* is None. *AuthenticationError* is raised if authentication fails. See *Authentication keys*.

class multiprocessing.connection.**Listener**([*address*[, *family*[, *backlog*[, *authkey*]]]])
A wrapper for a bound socket or Windows named pipe which is 'listening' for connections.

address is the address to be used by the bound socket or named pipe of the listener object.

Nota: If an address of '0.0.0.0' is used, the address will not be a connectable end point on Windows. If you require a connectable end-point, you should use '127.0.0.1'.

---

family is the type of socket (or named pipe) to use. This can be one of the strings `'AF_INET'` (for a TCP socket), `'AF_UNIX'` (for a Unix domain socket) or `'AF_PIPE'` (for a Windows named pipe). Of these only the first is guaranteed to be available. If *family* is `None` then the family is inferred from the format of *address*. If *address* is also `None` then a default is chosen. This default is the family which is assumed to be the fastest available. See

Formatos de Endereços. Note that if *family* is `'AF_UNIX'` and address is `None` then the socket will be created in a private temporary directory created using *tempfile.mkstemp()*.

If the listener object uses a socket then *backlog* (1 by default) is passed to the *listen()* method of the socket once it has been bound.

If *authkey* is given and not None, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if *authkey* is None. *AuthenticationError* is raised if authentication fails. See *Authentication keys*.

accept()
Accept a connection on the bound socket or named pipe of the listener object and return a *Connection* object. If authentication is attempted and fails, then *AuthenticationError* is raised.

close()
Close the bound socket or named pipe of the listener object. This is called automatically when the listener is garbage collected. However it is advisable to call it explicitly.

Listener objects have the following read-only properties:

address
The address which is being used by the Listener object.

last_accepted
The address from which the last accepted connection came. If this is unavailable then it is `None`.

Novo na versão 3.3: Listener objects now support the context management protocol – see *Tipos de Gerenciador de Contexto*. *__enter__()* returns the listener object, and *__exit__()* calls *close()*.

multiprocessing.connection.**wait**(*object_list*, *timeout=None*)
Wait till an object in *object_list* is ready. Returns the list of those objects in *object_list* which are ready. If *timeout* is a float then the call blocks for at most that many seconds. If *timeout* is `None` then it will block for an unlimited period. A negative timeout is equivalent to a zero timeout.

For both Unix and Windows, an object can appear in *object_list* if it is

- a readable *Connection* object;
- a connected and readable *socket.socket* object; or
- the *sentinel* attribute of a *Process* object.

A connection or socket object is ready when there is data available to be read from it, or the other end has been closed.

Unix: `wait(object_list, timeout)` almost equivalent `select.select(object_list, [], [], timeout)`. The difference is that, if *select.select()* is interrupted by a signal, it can raise *OSError* with an error number of `EINTR`, whereas *wait()* will not.

Windows: An item in *object_list* must either be an integer handle which is waitable (according to the definition used by the documentation of the Win32 function `WaitForMultipleObjects()`) or it can be an object with a `fileno()` method which returns a socket handle or pipe handle. (Note that pipe handles and socket handles are **not** waitable handles.)

Novo na versão 3.3.

Examples

The following server code creates a listener which uses `'secret password'` as an authentication key. It then waits for a connection and sends some data to the client:

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)     # family is deduced to be 'AF_INET'

with Listener(address, authkey=b'secret password') as listener:
    with listener.accept() as conn:
        print('connection accepted from', listener.last_accepted)

        conn.send([2.25, None, 'junk', float])

        conn.send_bytes(b'hello')

        conn.send_bytes(array('i', [42, 1729]))
```

The following code connects to the server and receives some data from the server:

```
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)

with Client(address, authkey=b'secret password') as conn:
    print(conn.recv())                  # => [2.25, None, 'junk', float]

    print(conn.recv_bytes())            # => 'hello'

    arr = array('i', [0, 0, 0, 0, 0])
    print(conn.recv_bytes_into(arr))    # => 8
    print(arr)                          # => array('i', [42, 1729, 0, 0, 0])
```

The following code uses *wait()* to wait for messages from multiple processes at once:

```
import time, random
from multiprocessing import Process, Pipe, current_process
from multiprocessing.connection import wait

def foo(w):
    for i in range(10):
        w.send((i, current_process().name))
    w.close()

if __name__ == '__main__':
    readers = []

    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # We close the writable end of the pipe now to be sure that
        # p is the only process which owns a handle for it.  This
        # ensures that when p closes its handle for the writable end,
        # wait() will promptly report the readable end as being ready.
        w.close()

    while readers:
```

(continua na próxima página)

(continuação da página anterior)

```
        for r in wait(readers):
            try:
                msg = r.recv()
            except EOFError:
                readers.remove(r)
            else:
                print(msg)
```

Formatos de Endereços

- Um endereço `'AF_INET'` é uma tupla na forma de `(hostname, port)` sendo *hostname* uma string e *port* um inteiro.
- An `'AF_UNIX'` address is a string representing a filename on the filesystem.
- **An `'AF_PIPE'` address is a string of the form** `r'\.\pipe{PipeName}'`. To use *`Client()`* to connect to a named pipe on a remote computer called *ServerName* one should use an address of the form `r'\`*`ServerName`*`\pipe{PipeName}'` instead.

Note that any string beginning with two backslashes is assumed by default to be an `'AF_PIPE'` address rather than an `'AF_UNIX'` address.

Authentication keys

When one uses *`Connection.recv`*, the data received is automatically unpickled. Unfortunately unpickling data from an untrusted source is a security risk. Therefore *`Listener`* and *`Client()`* use the *`hmac`* module to provide digest authentication.

An authentication key is a byte string which can be thought of as a password: once a connection is established both ends will demand proof that the other knows the authentication key. (Demonstrating that both ends are using the same key does **not** involve sending the key over the connection.)

If authentication is requested but no authentication key is specified then the return value of `current_process().authkey` is used (see *`Process`*). This value will be automatically inherited by any *`Process`* object that the current process creates. This means that (by default) all processes of a multi-process program will share a single authentication key which can be used when setting up connections between themselves.

Suitable authentication keys can also be generated by using *`os.urandom()`*.

Gerando logs

Some support for logging is available. Note, however, that the *`logging`* package does not use process shared locks so it is possible (depending on the handler type) for messages from different processes to get mixed up.

`multiprocessing.`**`get_logger`**`()`

Returns the logger used by *`multiprocessing`*. If necessary, a new one will be created.

When first created the logger has level `logging.NOTSET` and no default handler. Messages sent to this logger will not by default propagate to the root logger.

Note that on Windows child processes will only inherit the level of the parent process's logger – any other customization of the logger will not be inherited.

multiprocessing.**log_to_stderr**()
: This function performs a call to *get_logger()* but in addition to returning the logger created by get_logger, it adds a handler which sends output to *sys.stderr* using format `'[%(levelname)s/%(processName)s] %(message)s'`.

Below is an example session with logging turned on:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

For a full table of logging levels, see the *logging* module.

The `multiprocessing.dummy` module

multiprocessing.dummy replicates the API of *multiprocessing* but is no more than a wrapper around the *threading* module.

In particular, the `Pool` function provided by *multiprocessing.dummy* returns an instance of *ThreadPool*, which is a subclass of *Pool* that supports all the same method calls but uses a pool of worker threads rather than worker processes.

class multiprocessing.pool.**ThreadPool**([*processes*[, *initializer*[, *initargs*]]])
: A thread pool object which controls a pool of worker threads to which jobs can be submitted. *ThreadPool* instances are fully interface compatible with *Pool* instances, and their resources must also be properly managed, either by using the pool as a context manager or by calling *close()* and *terminate()* manually.

  processes is the number of worker threads to use. If *processes* is `None` then the number returned by *os.cpu_count()* is used.

  If *initializer* is not `None` then each worker process will call `initializer(*initargs)` when it starts.

  Unlike *Pool*, *maxtasksperchild* and *context* cannot be provided.

  > **Nota:** A *ThreadPool* shares the same interface as *Pool*, which is designed around a pool of processes and predates the introduction of the *concurrent.futures* module. As such, it inherits some operations that don't make sense for a pool backed by threads, and it has its own type for representing the status of asynchronous jobs, *AsyncResult*, that is not understood by any other libraries.
  >
  > Users should generally prefer to use *concurrent.futures.ThreadPoolExecutor*, which has a simpler interface that was designed around threads from the start, and which returns *concurrent.futures.Future* instances that are compatible with many other libraries, including *asyncio*.

17.2.3 Programming guidelines

There are certain guidelines and idioms which should be adhered to when using `multiprocessing`.

All start methods

The following applies to all start methods.

Avoid shared state

> As far as possible one should try to avoid shifting large amounts of data between processes.
>
> It is probably best to stick to using queues or pipes for communication between processes rather than using the lower level synchronization primitives.

Picklability

> Ensure that the arguments to the methods of proxies are picklable.

Thread safety of proxies

> Do not use a proxy object from more than one thread unless you protect it with a lock.
>
> (There is never a problem with different processes using the *same* proxy.)

Joining zombie processes

> On Unix when a process finishes but has not been joined it becomes a zombie. There should never be very many because each time a new process starts (or `active_children()` is called) all completed processes which have not yet been joined will be joined. Also calling a finished process's `Process.is_alive` will join the process. Even so it is probably good practice to explicitly join all the processes that you start.

Better to inherit than pickle/unpickle

> When using the *spawn* or *forkserver* start methods many types from `multiprocessing` need to be picklable so that child processes can use them. However, one should generally avoid sending shared objects to other processes using pipes or queues. Instead you should arrange the program so that a process which needs access to a shared resource created elsewhere can inherit it from an ancestor process.

Avoid terminating processes

> Using the `Process.terminate` method to stop a process is liable to cause any shared resources (such as locks, semaphores, pipes and queues) currently being used by the process to become broken or unavailable to other processes.
>
> Therefore it is probably best to only consider using `Process.terminate` on processes which never use any shared resources.

Joining processes that use queues

> Bear in mind that a process that has put items in a queue will wait before terminating until all the buffered items are fed by the "feeder" thread to the underlying pipe. (The child process can call the `Queue.cancel_join_thread` method of the queue to avoid this behaviour.)
>
> This means that whenever you use a queue you need to make sure that all items which have been put on the queue will eventually be removed before the process is joined. Otherwise you cannot be sure that processes which have put items on the queue will terminate. Remember also that non-daemonic processes will be joined automatically.
>
> An example which will deadlock is the following:

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()                    # this deadlocks
    obj = queue.get()
```

A fix here would be to swap the last two lines (or simply remove the `p.join()` line).

Explicitly pass resources to child processes

On Unix using the *fork* start method, a child process can make use of a shared resource created in a parent process using a global resource. However, it is better to pass the object as an argument to the constructor for the child process.

Apart from making the code (potentially) compatible with Windows and the other start methods this also ensures that as long as the child process is still alive the object will not be garbage collected in the parent process. This might be important if some resource is freed when the object is garbage collected in the parent process.

So for instance

```
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

should be rewritten as

```
from multiprocessing import Process, Lock

def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

Beware of replacing *sys.stdin* with a "file like object"

multiprocessing originally unconditionally called:

```
os.close(sys.stdin.fileno())
```

in the `multiprocessing.Process._bootstrap()` method — this resulted in issues with processes-in-processes. This has been changed to:

```
sys.stdin.close()
sys.stdin = open(os.open(os.devnull, os.O_RDONLY), closefd=False)
```

Which solves the fundamental issue of processes colliding with each other resulting in a bad file descriptor error, but introduces a potential danger to applications which replace *sys.stdin()* with a "file-like object" with output buffering. This danger is that if multiple processes call *close()* on this file-like object, it could result in the same data being flushed to the object multiple times, resulting in corruption.

If you write a file-like object and implement your own caching, you can make it fork-safe by storing the pid whenever you append to the cache, and discarding the cache when the pid changes. For example:

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

For more information, see bpo-5155, bpo-5313 and bpo-5331

The *spawn* and *forkserver* start methods

There are a few extra restriction which don't apply to the *fork* start method.

More picklability

Ensure that all arguments to `Process.__init__()` are picklable. Also, if you subclass *Process* then make sure that instances will be picklable when the *Process.start* method is called.

Global variables

Bear in mind that if code run in a child process tries to access a global variable, then the value it sees (if any) may not be the same as the value in the parent process at the time that *Process.start* was called.

However, global variables which are just module level constants cause no problems.

Safe importing of main module

Make sure that the main module can be safely imported by a new Python interpreter without causing unintended side effects (such a starting a new process).

For example, using the *spawn* or *forkserver* start method running the following module would fail with a *RuntimeError*:

```
from multiprocessing import Process

def foo():
    print('hello')

p = Process(target=foo)
p.start()
```

Instead one should protect the "entry point" of the program by using `if __name__ == '__main__':` as follows:

```
from multiprocessing import Process, freeze_support, set_start_method

```

(continua na próxima página)

(continuação da página anterior)

```
def foo():
    print('hello')

if __name__ == '__main__':
    freeze_support()
    set_start_method('spawn')
    p = Process(target=foo)
    p.start()
```

(The `freeze_support()` line can be omitted if the program will be run normally instead of frozen.)

This allows the newly spawned Python interpreter to safely import the module and then run the module's `foo()` function.

Similar restrictions apply if a pool or manager is created in the main module.

17.2.4 Exemplos

Demonstration of how to create and use customized managers and proxies:

```
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo:
    def f(self):
        print('you called Foo.f()')
    def g(self):
        print('you called Foo.g()')
    def _h(self):
        print('you called Foo._h()')

# A simple generator function
def baz():
    for i in range(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ['__next__']
    def __iter__(self):
        return self
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass
```

(continua na próxima página)

(continuação da página anterior)

```
# register the Foo class; make `f()` and `g()` accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)

##

def test():
    manager = MyManager()
    manager.start()

    print('-' * 20)

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print('-' * 20)

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print('-' * 20)

    it = manager.baz()
    for i in it:
        print('<%d>' % i, end=' ')
    print()

    print('-' * 20)

    op = manager.operator()
    print('op.add(23, 45) =', op.add(23, 45))
    print('op.pow(2, 94) =', op.pow(2, 94))
    print('op._exposed_ =', op._exposed_)

##

if __name__ == '__main__':
    freeze_support()
    test()
```

Using *Pool*:

```
import multiprocessing
import time
import random
import sys

#
# Functions used by test code
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
        )

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

def f(x):
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

#
# Test code
#

def test():
    PROCESSES = 4
    print('Creating pool with %d processes\n' % PROCESSES)

    with multiprocessing.Pool(PROCESSES) as pool:
        #
        # Tests
        #

        TASKS = [(mul, (i, 7)) for i in range(10)] + \
                [(plus, (i, 8)) for i in range(10)]

        results = [pool.apply_async(calculate, t) for t in TASKS]
        imap_it = pool.imap(calculatestar, TASKS)
        imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

        print('Ordered results using pool.apply_async():')
```

(continua na próxima página)

(continuação da página anterior)

```
        for r in results:
            print('\t', r.get())
        print()

        print('Ordered results using pool.imap():')
        for x in imap_it:
            print('\t', x)
        print()

        print('Unordered results using pool.imap_unordered():')
        for x in imap_unordered_it:
            print('\t', x)
        print()

        print('Ordered results using pool.map() --- will block till complete:')
        for x in pool.map(calculatestar, TASKS):
            print('\t', x)
        print()

        #
        # Test error handling
        #

        print('Testing error handling:')

        try:
            print(pool.apply(f, (5,)))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.apply()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(pool.map(f, list(range(10))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.map()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(list(pool.imap(f, list(range(10)))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from list(pool.imap())')
        else:
            raise AssertionError('expected ZeroDivisionError')

        it = pool.imap(f, list(range(10)))
        for i in range(10):
            try:
                x = next(it)
            except ZeroDivisionError:
                if i == 5:
                    pass
            except StopIteration:
                break
            else:
                if i == 5:
```

(continua na próxima página)

(continuação da página anterior)

```
                    raise AssertionError('expected ZeroDivisionError')

        assert i == 9
        print('\tGot ZeroDivisionError as expected from IMapIterator.next()')
        print()

        #
        # Testing timeouts
        #

        print('Testing ApplyResult.get() with timeout:', end=' ')
        res = pool.apply_async(calculate, TASKS[0])
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % res.get(0.02))
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()

        print('Testing IMapIterator.next() with timeout:', end=' ')
        it = pool.imap(calculatestar, TASKS)
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % it.next(0.02))
            except StopIteration:
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()


if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()
```

An example showing how to use queues to feed tasks to a collection of worker processes and collect the results:

```
import time
import random

from multiprocessing import Process, Queue, current_process, freeze_support

#
# Function run by worker processes
#

def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

#
```

(continua na próxima página)

(continuação da página anterior)

```
# Function used to calculate result
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)

#
# Functions referenced by tasks
#

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
    print('Unordered results:')
    for i in range(len(TASKS1)):
        print('\t', done_queue.get())

    # Add more tasks using `put()`
    for task in TASKS2:
        task_queue.put(task)

    # Get and print some more results
    for i in range(len(TASKS2)):
        print('\t', done_queue.get())

    # Tell child processes to stop
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')
```

(continua na próxima página)

(continuação da página anterior)

```
if __name__ == '__main__':
    freeze_support()
    test()
```

17.3 O pacote `concurrent`

Atualmente, há apenas um módulo neste pacote:

- *`concurrent.futures`* – Iniciando tarefas em paralelo

17.4 `concurrent.futures` — Iniciando tarefas em paralelo

Novo na versão 3.2.

Código-fonte: Lib/concurrent/futures/thread.py e Lib/concurrent/futures/process.py

---

The *`concurrent.futures`* module provides a high-level interface for asynchronously executing callables.

The asynchronous execution can be performed with threads, using *`ThreadPoolExecutor`*, or separate processes, using *`ProcessPoolExecutor`*. Both implement the same interface, which is defined by the abstract *`Executor`* class.

17.4.1 Executor Objects

class concurrent.futures.**Executor**

An abstract class that provides methods to execute calls asynchronously. It should not be used directly, but through its concrete subclasses.

submit(*fn*, **args*, ***kwargs*)

Schedules the callable, *fn*, to be executed as `fn(*args **kwargs)` and returns a *`Future`* object representing the execution of the callable.

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

map(*func*, **iterables*, *timeout=None*, *chunksize=1*)

Similar to *`map(func, *iterables)`* except:

- the *iterables* are collected immediately rather than lazily;
- *func* is executed asynchronously and several calls to *func* may be made concurrently.

The returned iterator raises a *`concurrent.futures.TimeoutError`* if *`__next__()`* is called and the result isn't available after *timeout* seconds from the original call to *`Executor.map()`*. *timeout* can be an int or a float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If a *func* call raises an exception, then that exception will be raised when its value is retrieved from the iterator.

When using *ProcessPoolExecutor*, this method chops *iterables* into a number of chunks which it submits to the pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer. For very long iterables, using a large value for *chunksize* can significantly improve performance compared to the default size of 1. With *ThreadPoolExecutor*, *chunksize* has no effect.

Alterado na versão 3.5: Adicionado o argumento *chunksize*.

shutdown(*wait=True*)

Signal the executor that it should free any resources that it is using when the currently pending futures are done executing. Calls to *Executor.submit()* and *Executor.map()* made after shutdown will raise *RuntimeError*.

If *wait* is `True` then this method will not return until all the pending futures are done executing and the resources associated with the executor have been freed. If *wait* is `False` then this method will return immediately and the resources associated with the executor will be freed when all pending futures are done executing. Regardless of the value of *wait*, the entire Python program will not exit until all pending futures are done executing.

You can avoid having to call this method explicitly if you use the `with` statement, which will shutdown the *Executor* (waiting as if *Executor.shutdown()* were called with *wait* set to `True`):

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

17.4.2 ThreadPoolExecutor

ThreadPoolExecutor is an *Executor* subclass that uses a pool of threads to execute calls asynchronously.

Deadlocks can occur when the callable associated with a *Future* waits on the results of another *Future*. For example:

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result())  # b will never complete because it is waiting on a.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result())  # a will never complete because it is waiting on b.
    return 6


executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

And:

```
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker thread and
```

(continua na próxima página)

(continuação da página anterior)

```
    # it is executing this function.
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

class concurrent.futures.**ThreadPoolExecutor**(*max_workers=None*, *thread_name_prefix=''*, *initializer=None*, *initargs=()*)

An *Executor* subclass that uses a pool of at most *max_workers* threads to execute calls asynchronously.

initializer is an optional callable that is called at the start of each worker thread; *initargs* is a tuple of arguments passed to the initializer. Should *initializer* raise an exception, all currently pending jobs will raise a *BrokenThreadPool*, as well as any attempt to submit more jobs to the pool.

Alterado na versão 3.5: If *max_workers* is `None` or not given, it will default to the number of processors on the machine, multiplied by `5`, assuming that *ThreadPoolExecutor* is often used to overlap I/O instead of CPU work and the number of workers should be higher than the number of workers for *ProcessPoolExecutor*.

Novo na versão 3.6: The *thread_name_prefix* argument was added to allow users to control the *threading.Thread* names for worker threads created by the pool for easier debugging.

Alterado na versão 3.7: Added the *initializer* and *initargs* arguments.

Exemplo de ThreadPoolExecutor

```
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://nonexistant-subdomain.python.org/']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# We can use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # Start the load operations and mark each future with its URL
    future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))
```

17.4.3 `ProcessPoolExecutor`

The *ProcessPoolExecutor* class is an *Executor* subclass that uses a pool of processes to execute calls asynchronously. *ProcessPoolExecutor* uses the *multiprocessing* module, which allows it to side-step the *Global Interpreter Lock* but also means that only picklable objects can be executed and returned.

The __main__ module must be importable by worker subprocesses. This means that *ProcessPoolExecutor* will not work in the interactive interpreter.

Calling *Executor* or *Future* methods from a callable submitted to a *ProcessPoolExecutor* will result in deadlock.

class concurrent.futures.**ProcessPoolExecutor**(*max_workers=None*, *mp_context=None*, *initializer=None*, *initargs=()*)

An *Executor* subclass that executes calls asynchronously using a pool of at most *max_workers* processes. If *max_workers* is None or not given, it will default to the number of processors on the machine. If *max_workers* is lower or equal to 0, then a *ValueError* will be raised. On Windows, *max_workers* must be equal or lower than 61. If it is not then *ValueError* will be raised. If *max_workers* is None, then the default chosen will be at most 61, even if more processors are available. *mp_context* can be a multiprocessing context or None. It will be used to launch the workers. If *mp_context* is None or not given, the default multiprocessing context is used.

initializer is an optional callable that is called at the start of each worker process; *initargs* is a tuple of arguments passed to the initializer. Should *initializer* raise an exception, all currently pending jobs will raise a *BrokenProcessPool*, as well any attempt to submit more jobs to the pool.

Alterado na versão 3.3: When one of the worker processes terminates abruptly, a BrokenProcessPool error is now raised. Previously, behaviour was undefined but operations on the executor or its futures would often freeze or deadlock.

Alterado na versão 3.7: The *mp_context* argument was added to allow users to control the start_method for worker processes created by the pool.

Added the *initializer* and *initargs* arguments.

ProcessPoolExecutor Example

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True
```

(continua na próxima página)

(continuação da página anterior)

```
def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))

if __name__ == '__main__':
    main()
```

17.4.4 Future Objects

The *Future* class encapsulates the asynchronous execution of a callable. *Future* instances are created by *Executor.submit()*.

class concurrent.futures.**Future**

Encapsulates the asynchronous execution of a callable. *Future* instances are created by *Executor.submit()* and should not be created directly except for testing.

cancel()

Attempt to cancel the call. If the call is currently being executed or finished running and cannot be cancelled then the method will return `False`, otherwise the call will be cancelled and the method will return `True`.

cancelled()

Return `True` if the call was successfully cancelled.

running()

Return `True` if the call is currently being executed and cannot be cancelled.

done()

Return `True` if the call was successfully cancelled or finished running.

result(*timeout=None*)

Return the value returned by the call. If the call hasn't yet completed then this method will wait up to *timeout* seconds. If the call hasn't completed in *timeout* seconds, then a *concurrent.futures.TimeoutError* will be raised. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If the future is cancelled before completing then *CancelledError* will be raised.

If the call raised, this method will raise the same exception.

exception(*timeout=None*)

Return the exception raised by the call. If the call hasn't yet completed then this method will wait up to *timeout* seconds. If the call hasn't completed in *timeout* seconds, then a *concurrent.futures.TimeoutError* will be raised. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If the future is cancelled before completing then *CancelledError* will be raised.

If the call completed without raising, `None` is returned.

add_done_callback(*fn*)

Attaches the callable *fn* to the future. *fn* will be called, with the future as its only argument, when the future is cancelled or finishes running.

Added callables are called in the order that they were added and are always called in a thread belonging to the process that added them. If the callable raises an *Exception* subclass, it will be logged and ignored. If the callable raises a *BaseException* subclass, the behavior is undefined.

If the future has already completed or been cancelled, *fn* will be called immediately.

The following `Future` methods are meant for use in unit tests and `Executor` implementations.

set_running_or_notify_cancel()
This method should only be called by `Executor` implementations before executing the work associated with the `Future` and by unit tests.

If the method returns `False` then the `Future` was cancelled, i.e. `Future.cancel()` was called and returned *True*. Any threads waiting on the `Future` completing (i.e. through `as_completed()` or `wait()`) will be woken up.

If the method returns `True` then the `Future` was not cancelled and has been put in the running state, i.e. calls to `Future.running()` will return *True*.

This method can only be called once and cannot be called after `Future.set_result()` or `Future.set_exception()` have been called.

set_result(*result*)
Sets the result of the work associated with the `Future` to *result*.

This method should only be used by `Executor` implementations and unit tests.

set_exception(*exception*)
Sets the result of the work associated with the `Future` to the `Exception` *exception*.

This method should only be used by `Executor` implementations and unit tests.

17.4.5 Module Functions

concurrent.futures.**wait**(*fs*, *timeout=None*, *return_when=ALL_COMPLETED*)
Wait for the `Future` instances (possibly created by different `Executor` instances) given by *fs* to complete. Returns a named 2-tuple of sets. The first set, named `done`, contains the futures that completed (finished or cancelled futures) before the wait completed. The second set, named `not_done`, contains the futures that did not complete (pending or running futures).

timeout can be used to control the maximum number of seconds to wait before returning. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

return_when indica quando esta função deve retornar. Ele deve ser uma das seguintes constantes:

Constante	Description (descrição)
`FIRST_COMPLETED`	A função irá retornar quando qualquer futuro terminar ou for cancelado.
`FIRST_EXCEPTION`	A função irá retornar quando qualquer futuro encerrar levantando uma exceção. Se nenhum futuro levantar uma exceção, então é equivalente a `ALL_COMPLETED`.
`ALL_COMPLETED`	A função irá retornar quando todos os futuros encerrarem ou forem cancelados.

concurrent.futures.**as_completed**(*fs*, *timeout=None*)
Returns an iterator over the `Future` instances (possibly created by different `Executor` instances) given by *fs* that yields futures as they complete (finished or cancelled futures). Any futures given by *fs* that are duplicated will be returned once. Any futures that completed before `as_completed()` is called will be yielded first. The returned iterator raises a `concurrent.futures.TimeoutError` if `__next__()` is called and the result isn't available after *timeout* seconds from the original call to `as_completed()`. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

Ver também:

PEP 3148 – futures - execute computations asynchronously The proposal which described this feature for inclusion in the Python standard library.

17.4.6 Exception classes

exception concurrent.futures.**CancelledError**
Raised when a future is cancelled.

exception concurrent.futures.**TimeoutError**
Raised when a future operation exceeds the given timeout.

exception concurrent.futures.**BrokenExecutor**
Derived from *RuntimeError*, this exception class is raised when an executor is broken for some reason, and cannot be used to submit or execute new tasks.

Novo na versão 3.7.

exception concurrent.futures.thread.**BrokenThreadPool**
Derived from *BrokenExecutor*, this exception class is raised when one of the workers of a ThreadPoolExecutor has failed initializing.

Novo na versão 3.7.

exception concurrent.futures.process.**BrokenProcessPool**
Derived from *BrokenExecutor* (formerly *RuntimeError*), this exception class is raised when one of the workers of a ProcessPoolExecutor has terminated in a non-clean fashion (for example, if it was killed from the outside).

Novo na versão 3.3.

17.5 subprocess — Gerenciamento de subprocessos

Código Fonte: Lib/subprocess.py

O módulo *subprocess* permite que você crie novos processos, conecte-se aos seus canais de entrada/saída/erro e obtenha seus códigos de retorno. Este módulo pretende substituir vários módulos e funções mais antigos:

```
os.system
os.spawn*
```

Informações sobre como o módulo *subprocess* pode ser usado para substituir esses módulos e funções podem ser encontradas nas seções a seguir.

Ver também:

PEP 324 – PEP propondo o módulo subprocess

17.5.1 Usando o módulo subprocess

A abordagem recomendada para invocar subprocessos é usar a função *run()* para todos os casos de uso que ela pode manipular. Para casos de uso mais avançados, a interface subjacente *Popen* pode ser usada diretamente.

A função *run()* foi adicionada no Python 3.5; se você precisa manter a compatibilidade com versões anteriores, veja a seção *API de alto nível mais antiga*.

subprocess.**run**(*args*, ***, *stdin=None*, *input=None*, *stdout=None*, *stderr=None*, *capture_output=False*, *shell=False*, *cwd=None*, *timeout=None*, *check=False*, *encoding=None*, *errors=None*, *text=None*, *env=None*, *universal_newlines=None*, ***other_popen_kwargs*)
Executa o comando descrito por *args*. Aguarda a conclusão do comando e retorna uma instância de *CompletedProcess*.

Os argumentos mostrados acima são meramente os mais comuns, descritos abaixo em *Argumentos Usados Freqüentemente* (daí o uso de notação somente-nomeado na assinatura abreviada). A assinatura da função completa é basicamente a mesma do construtor `Popen` - a maioria dos argumentos para esta função são passados para aquela interface. (*timeout*, *input*, *check* e *capture_output* não são.)

Se *capture_output* for verdadeiro, stdout e stderr serão capturados. Quando usado, o objeto interno `Popen` é automaticamente criado com `stdout=PIPE` e `stderr=PIPE`. Os argumentos *stdout * e *stderr* não podem ser fornecidos ao mesmo tempo que *capture_output*. Se você deseja capturar e combinar os dois fluxos em um, use `stdout=PIPE` e `stderr=STDOUT` ao invés de *capture_output*.

O argumento *timeout* é passado para `Popen.communicate()`. Se o tempo limite expirar, o processo filho será encerrado e aguardado. A exceção `TimeoutExpired` será levantada novamente após o término do processo filho.

O argumento *input* é passado para `Popen.communicate()` e, portanto, para o stdin do subprocesso. Se usado, deve ser uma sequência de bytes ou uma string se *encoding* ou *errors* forem especificados ou *text* for verdadeiro. Quando usado, o objeto internal `Popen` é automaticamente criado com `stdin=PIPE`, e o argumento *stdin* também não pode ser usado.

Se *check* for verdadeiro, e o processo sair com um código de saída diferente de zero, uma exceção `CalledProcessError` será levantada. Os atributos dessa exceção contêm os argumentos, o código de saída e stdout e stderr se eles foram capturados.

Se *encoding* ou *errors* forem especificados, ou *text* for verdadeiro, os objetos de arquivo para stdin, stdout e stderr são abertos em modo de texto usando a *encoding* e *errors* especificados ou o `io.TextIOWrapper` padrão. O argumento *universal_newlines* é equivalente a *text* e é fornecido para compatibilidade com versões anteriores. Por padrão, os objetos arquivo são abertos no modo binário.

Se *env* não for `None`, deve ser um mapeamento que define as variáveis de ambiente para o novo processo; eles são usados em vez do comportamento padrão de herdar o ambiente do processo atual. É passado diretamente para `Popen`.

Exemplos:

```
>>> subprocess.run(["ls", "-l"])  # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], capture_output=True)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n', stderr=b'')
```

Novo na versão 3.5.

Alterado na versão 3.6: Adicionado os parâmetros *encoding* e *errors*.

Alterado na versão 3.7: Adicionado o parâmetro *text*, como um apelido mais compreensível que *universal_newlines*. Adicionado o parâmetro *capture_output*.

Alterado na versão 3.7.17: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

class subprocess.**CompletedProcess**

O valor de retorno de `run()`, representando um processo que foi concluído.

args
Os argumentos usados para iniciar o processo. Pode ser uma lista ou uma string.

returncode
Status de saída do processo filho. Normalmente, um status de saída 0 indica que foi executado com êxito.

Um valor negativo `-N` indica que o filho foi terminado pelo sinal `N` (POSIX apenas).

stdout
Stdout capturado do processo filho. Uma sequência de bytes ou uma string se *run()* foi chamada com uma codificação, erros ou text=True. `None` se stdout não foi capturado.

Se você executou o processo com `stderr=subprocess.STDOUT`, stdout e stderr serão combinados neste atributo, e *stderr* será `None`.

stderr
Stderr capturado do processo filho. Uma sequência de bytes ou uma string se *run()* foi chamada com uma codificação, erros ou text=True. `None` se stderr não foi capturado.

check_returncode()
Se *returncode* é diferente de zero, levantar um *CalledProcessError*.

Novo na versão 3.5.

subprocess.**DEVNULL**
Valor especial que pode ser usado como o argumento *stdin*, *stdout* ou *stderr* para *Popen* e indica que o arquivo especial *os.devnull* será usado.

Novo na versão 3.3.

subprocess.**PIPE**
Valor especial que pode ser usado como o argumento *stdin*, *stdout* ou *stderr* para *Popen* e indica que um encadeamento para o fluxo padrão deve ser aberto. Mais útil com *Popen.communicate()*.

subprocess.**STDOUT**
Valor especial que pode ser usado como o argumento *stderr* para *Popen* e indica que o erro padrão deve ir para o mesmo manipulador que a saída padrão.

exception subprocess.**SubprocessError**
Classe base para todas as outras exceções deste módulo.

Novo na versão 3.3.

exception subprocess.**TimeoutExpired**
Subclasse de *SubprocessError*, levantada quando um tempo limite expira enquanto espera por um processo filho.

cmd
Comando que foi usado para gerar o processo filho.

timeout
Tempo limite em segundos.

output
Saída do processo filho se ele foi capturado por *run()* ou *check_output()*. Caso contrário, `None`.

stdout
Apelido para a saída, para simetria com *stderr*.

stderr
Saída Stderr do processo filho se ele foi capturado por *run()*. Caso contrário, `None`.

Novo na versão 3.3.

Alterado na versão 3.5: Atributos *stdout* e *stderr* adicionados

exception subprocess.**CalledProcessError**
Subclasse de *SubprocessError*, levantada quando um processo executado por *check_call()* ou *check_output()* retorna um status de saída diferente de zero.

returncode
Status de saída do processo filho. Se o processo foi encerrado devido a um sinal, este será o número do sinal negativo.

cmd
Comando que foi usado para gerar o processo filho.

output
Saída do processo filho se ele foi capturado por *run()* ou *check_output()*. Caso contrário, None.

stdout
Apelido para a saída, para simetria com *stderr*.

stderr
Saída Stderr do processo filho se ele foi capturado por *run()*. Caso contrário, None.

Alterado na versão 3.5: Atributos *stdout* e *stderr* adicionados

Argumentos Usados Freqüentemente

Para suportar uma ampla variedade de casos de uso, o construtor *Popen* (e as funções de conveniência) aceita muitos argumentos opcionais. Para a maioria dos casos de uso típicos, muitos desses argumentos podem ser seguramente deixados em seus valores padrão. Os argumentos mais comumente necessários são:

args é necessário para todas as chamadas e deve ser uma string ou uma sequência de argumentos do programa. Fornecer uma sequência de argumentos geralmente é preferível, pois permite que o módulo cuide de qualquer escapamento e citação de argumentos que forem necessários (por exemplo, para permitir espaços em nomes de arquivo). Se for passada uma única string, *shell* deve ser *True* (veja abaixo) ou então a string deve apenas nomear o programa a ser executado sem especificar nenhum argumento.

stdin, *stdout* e *stderr* especificam o manipulador de arquivo de entrada padrão, a saída padrão e de erro padrão do programa executado, respectivamente. Os valores válidos são *PIPE*, *DEVNULL*, um descritor de arquivo existente (um inteiro positivo), um objeto arquivo existente e None. *PIPE* indica que um novo encadeamento para o filho deve ser criado. *DEVNULL* indica que o arquivo especial *os.devnull* será usado. Com as configurações padrão de None, nenhum redirecionamento ocorrerá; os manipuladores de arquivo do filho serão herdados do pai. Além disso, *stderr* pode ser *STDOUT*, o que indica que os dados stderr do processo filho devem ser capturados no mesmo manipulador de arquivo que para *stdout*.

Se *encoding* ou *errors* forem especificados, ou *text* (também conhecido como *universal_newlines*) for verdadeiro, os objetos de arquivo *stdin*, *stdout* e *stderr* serão abertos em modo de texto usando a *encoding* e *errors* especificados na chamada ou os valores padrão para *io.TextIOWrapper*.

Para *stdin*, os caracteres de fim de linha '\n' na entrada serão convertidos para o separador de linha padrão *os.linesep*. Para *stdout* e *stderr*, todas as terminações de linha na saída serão convertidas para '\n'. Para obter mais informações, consulte a documentação da classe *io.TextIOWrapper* quando o argumento *newline* para seu construtor é None.

Se o modo de texto não for usado, *stdin*, *stdout* e *stderr* serão abertos como fluxos binários. Nenhuma codificação ou conversão de final de linha é executada.

Novo na versão 3.6: Adicionado os parâmetros *encoding* e *errors*.

Novo na versão 3.7: Adicionado o parâmetro *text* como um atalho para *universal_newlines*.

Nota: O atributo newlines dos objetos arquivo `Popen.stdin`, `Popen.stdout` e `Popen.stderr` não são atualizados pelo método `Popen.communicate()`.

Se *shell* for `True`, o comando especificado será executado através do shell. Isso pode ser útil se você estiver usando Python principalmente para o fluxo de controle aprimorado que ele oferece sobre a maioria dos shells do sistema e ainda deseja acesso conveniente a outros recursos do shell, como canais de shell, caracteres curinga de nome de arquivo, expansão de variável de ambiente e expansão de ~ para o diretório inicial de um usuário. No entanto, observe que o próprio Python oferece implementações de muitos recursos semelhantes a shell (em particular, `glob`, `fnmatch`, `os.walk()`, `os.path.expandvars()`, `os.path.expanduser()` e `shutil`).

Alterado na versão 3.3: Quando *universal_newlines* é `True`, a classe usa a codificação `locale.getpreferredencoding(False)` em vez de"locale.getpreferredencoding()". Veja a classe `io.TextIOWrapper` para mais informações sobre esta alteração.

Nota: Leia a seção *Security Considerations* antes de usar `shell=True`.

Essas opções, junto com todas as outras opções, são descritas em mais detalhes na documentação do construtor `Popen`.

Construtor Popen

A criação e gerenciamento do processo subjacente neste módulo é manipulado pela classe `Popen`. Ela oferece muita flexibilidade para que os desenvolvedores sejam capazes de lidar com os casos menos comuns não cobertos pelas funções de conveniência.

class subprocess.**Popen**(*args*, *bufsize=-1*, *executable=None*, *stdin=None*, *stdout=None*, *stderr=None*, *preexec_fn=None*, *close_fds=True*, *shell=False*, *cwd=None*, *env=None*, *universal_newlines=None*, *startupinfo=None*, *creationflags=0*, *restore_signals=True*, *start_new_session=False*, *pass_fds=()*, ***, *encoding=None*, *errors=None*, *text=None*)

Executa um programa filho em um novo processo. No POSIX, a classe usa o comportamento de `os.execvp()` para executar o programa filho. No Windows, a classe usa a função Windows `CreateProcess()`. Os argumentos para `Popen` são os seguintes.

args should be a sequence of program arguments or else a single string. By default, the program to execute is the first item in *args* if *args* is a sequence. If *args* is a string, the interpretation is platform-dependent and described below. See the *shell* and *executable* arguments for additional differences from the default behavior. Unless otherwise stated, it is recommended to pass *args* as a sequence.

Um exemplo de passagem de alguns argumentos para um programa externo como uma sequência é:

```
Popen(["/usr/bin/git", "commit", "-m", "Fixes a bug."])
```

No POSIX, se *args* for uma string, a string é interpretada como o nome ou caminho do programa a ser executado. No entanto, isso só pode ser feito se não forem passados argumentos para o programa.

Nota: Pode não ser óbvio como quebrar um comando shell em uma sequência de argumentos, especialmente em casos complexos. `shlex.split()` pode ilustrar como determinar a tokenização correta para *args*:

```
>>> import shlex, subprocess
>>> command_line = input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
```

(continua na próxima página)

(continuação da página anterior)

```
>>> args = shlex.split(command_line)
>>> print(args)
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd', "echo '
↪$MONEY'"]
>>> p = subprocess.Popen(args) # Success!
```

Observe em particular que as opções (como *-input*) e argumentos (como *eggs.txt*) que são separados por espaços em branco no shell vão em elementos de lista separados, enquanto os argumentos que precisam de aspas ou escape de contrabarra quando usados no shell (como nomes de arquivos contendo espaços ou o comando *echo* mostrado acima) são um único elemento de lista.

No Windows, se *args* for uma sequência, será convertido em uma string da maneira descrita em *Converter uma sequência de argumentos em uma string no Windows*. Isso ocorre porque o `CreateProcess()` subjacente opera em strings.

The *shell* argument (which defaults to `False`) specifies whether to use the shell as the program to execute. If *shell* is `True`, it is recommended to pass *args* as a string rather than as a sequence.

On POSIX with `shell=True`, the shell defaults to `/bin/sh`. If *args* is a string, the string specifies the command to execute through the shell. This means that the string must be formatted exactly as it would be when typed at the shell prompt. This includes, for example, quoting or backslash escaping filenames with spaces in them. If *args* is a sequence, the first item specifies the command string, and any additional items will be treated as additional arguments to the shell itself. That is to say, *Popen* does the equivalent of:

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

On Windows with `shell=True`, the `COMSPEC` environment variable specifies the default shell. The only time you need to specify `shell=True` on Windows is when the command you wish to execute is built into the shell (e.g. **`dir`** or **`copy`**). You do not need `shell=True` to run a batch file or console-based executable.

Nota: Leia a seção *Security Considerations* antes de usar `shell=True`.

---

bufsize will be supplied as the corresponding argument to the *open()* function when creating the stdin/stdout/stderr pipe file objects:

- `0` means unbuffered (read and write are one system call and can return short)
- `1` means line buffered (only usable if `universal_newlines=True` i.e., in a text mode)
- any other positive value means use a buffer of approximately that size
- negative bufsize (the default) means the system default of io.DEFAULT_BUFFER_SIZE will be used.

Alterado na versão 3.3.1: *bufsize* now defaults to -1 to enable buffering by default to match the behavior that most code expects. In versions prior to Python 3.2.4 and 3.3.1 it incorrectly defaulted to `0` which was unbuffered and allowed short reads. This was unintentional and did not match the behavior of Python 2 as most code expected.

The *executable* argument specifies a replacement program to execute. It is very seldom needed. When `shell=False`, *executable* replaces the program to execute specified by *args*. However, the original *args* is still passed to the program. Most programs treat the program specified by *args* as the command name, which can then be different from the program actually executed. On POSIX, the *args* name becomes the display name for the executable in utilities such as **`ps`**. If `shell=True`, on POSIX the *executable* argument specifies a replacement shell for the default `/bin/sh`.

Alterado na versão 3.6: *executable* parameter accepts a *path-like object* on POSIX.

Alterado na versão 3.7.17: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are *PIPE*, *DEVNULL*, an existing file descriptor (a positive integer), an existing *file object*, and `None`. *PIPE* indicates that a new pipe to the child should be created. *DEVNULL* indicates that the special file *os.devnull* will be used. With the default settings of `None`, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, *stderr* can be *STDOUT*, which indicates that the stderr data from the applications should be captured into the same file handle as for stdout.

If *preexec_fn* is set to a callable object, this object will be called in the child process just before the child is executed. (POSIX only)

Aviso: The *preexec_fn* parameter is not safe to use in the presence of threads in your application. The child process could deadlock before exec is called. If you must use it, keep it trivial! Minimize the number of libraries you call into.

Nota: If you need to modify the environment for the child use the *env* parameter rather than doing it in a *preexec_fn*. The *start_new_session* parameter can take the place of a previously common use of *preexec_fn* to call os.setsid() in the child.

If *close_fds* is true, all file descriptors except `0`, `1` and `2` will be closed before the child process is executed. Otherwise when *close_fds* is false, file descriptors obey their inheritable flag as described in *Herança de descritores de arquivo*.

On Windows, if *close_fds* is true then no handles will be inherited by the child process unless explicitly passed in the `handle_list` element of *STARTUPINFO.lpAttributeList*, or by standard handle redirection.

Alterado na versão 3.2: The default for *close_fds* was changed from *False* to what is described above.

Alterado na versão 3.7: On Windows the default for *close_fds* was changed from *False* to *True* when redirecting the standard handles. It's now possible to set *close_fds* to *True* when redirecting the standard handles.

pass_fds is an optional sequence of file descriptors to keep open between the parent and child. Providing any *pass_fds* forces *close_fds* to be *True*. (POSIX only)

Novo na versão 3.2: O parâmetro *pass_fds* foi adicionado.

If *cwd* is not `None`, the function changes the working directory to *cwd* before executing the child. *cwd* can be a *str* and *path-like* object. In particular, the function looks for *executable* (or for the first item in *args*) relative to *cwd* if the executable path is a relative path.

Alterado na versão 3.6: *cwd* parameter accepts a *path-like object*.

If *restore_signals* is true (the default) all signals that Python has set to SIG_IGN are restored to SIG_DFL in the child process before the exec. Currently this includes the SIGPIPE, SIGXFZ and SIGXFSZ signals. (POSIX only)

Alterado na versão 3.2: *restore_signals* foi adicionado.

If *start_new_session* is true the setsid() system call will be made in the child process prior to the execution of the subprocess. (POSIX only)

Alterado na versão 3.2: *start_new_session* foi adicionado.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process' environment.

Nota: If specified, *env* must provide any variables required for the program to execute. On Windows, in order to run a side-by-side assembly the specified *env* **must** include a valid `SystemRoot`.

If *encoding* or *errors* are specified, or *text* is true, the file objects *stdin*, *stdout* and *stderr* are opened in text mode with the specified encoding and *errors*, as described above in *Argumentos Usados Freqüentemente*. The *universal_newlines* argument is equivalent to *text* and is provided for backwards compatibility. By default, file objects are opened in binary mode.

Novo na versão 3.6: *encoding* e *errors* foram adicionados.

Novo na versão 3.7: *text* foi adicionado como um atalho mais legível para *universal_newlines*.

If given, *startupinfo* will be a *STARTUPINFO* object, which is passed to the underlying `CreateProcess` function. *creationflags*, if given, can be one or more of the following flags:

- *CREATE_NEW_CONSOLE*
- *CREATE_NEW_PROCESS_GROUP*
- *ABOVE_NORMAL_PRIORITY_CLASS*
- *BELOW_NORMAL_PRIORITY_CLASS*
- *HIGH_PRIORITY_CLASS*
- *IDLE_PRIORITY_CLASS*
- *NORMAL_PRIORITY_CLASS*
- *REALTIME_PRIORITY_CLASS*
- *CREATE_NO_WINDOW*
- *DETACHED_PROCESS*
- *CREATE_DEFAULT_ERROR_MODE*
- *CREATE_BREAKAWAY_FROM_JOB*

Popen objects are supported as context managers via the `with` statement: on exit, standard file descriptors are closed, and the process is waited for.

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

Alterado na versão 3.2: Adicionado suporte a gerenciador de contexto.

Alterado na versão 3.6: O destruidor Popen agora emite um aviso *ResourceWarning* se o processo filho ainda estiver em execução.

Exceções

Exceções levantadas no processo filho, antes que o novo programa comece a ser executado, serão levantadas novamente no pai.

A exceção mais comum levantada é *OSError*. Isso ocorre, por exemplo, ao tentar executar um arquivo inexistente. Os aplicativos devem se preparar para exceções *OSError*.

A exceção *ValueError* será levantada se *Popen* for chamado com argumentos inválidos.

check_call() e *check_output()* levantarão *CalledProcessError* se o processo chamado retornar um código de retorno diferente de zero.

Todas as funções e métodos que aceitam um parâmetro de *timeout*, como *call()* e *Popen.communicate()* levantarão *TimeoutExpired* se o tempo limite expirar antes de o processo terminar.

Todas as exceções definidas neste módulo herdam de *SubprocessError*.

Novo na versão 3.3: A classe base *SubprocessError* foi adicionada.

17.5.2 Considerações de Segurança

Unlike some other popen functions, this implementation will never implicitly call a system shell. This means that all characters, including shell metacharacters, can safely be passed to child processes. If the shell is invoked explicitly, via `shell=True`, it is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately to avoid shell injection vulnerabilities.

When using `shell=True`, the *shlex.quote()* function can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

17.5.3 Objetos Popen

Instâncias da classe *Popen* têm os seguintes métodos:

`Popen.`**`poll`**`()`
Check if child process has terminated. Set and return *returncode* attribute. Otherwise, returns `None`.

`Popen.`**`wait`**`(`*timeout=None*`)`
Wait for child process to terminate. Set and return *returncode* attribute.

If the process does not terminate after *timeout* seconds, raise a *TimeoutExpired* exception. It is safe to catch this exception and retry the wait.

Nota: This will deadlock when using `stdout=PIPE` or `stderr=PIPE` and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use *Popen.communicate()* when using pipes to avoid that.

Nota: The function is implemented using a busy loop (non-blocking call and short sleeps). Use the *asyncio* module for an asynchronous wait: see *asyncio.create_subprocess_exec*.

Alterado na versão 3.3: *timeout* foi adicionado.

`Popen.`**`communicate`**`(`*input=None*, *timeout=None*`)`
Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate. The optional *input* argument should be data to be sent to the child process, or `None`, if no data should be sent to the child. If streams were opened in text mode, *input* must be a string. Otherwise, it must be bytes.

communicate() returns a tuple `(stdout_data, stderr_data)`. The data will be strings if streams were opened in text mode; otherwise, bytes.

Note that if you want to send data to the process's stdin, you need to create the Popen object with `stdin=PIPE`. Similarly, to get anything other than `None` in the result tuple, you need to give `stdout=PIPE` and/or `stderr=PIPE` too.

If the process does not terminate after *timeout* seconds, a *TimeoutExpired* exception will be raised. Catching this exception and retrying communication will not lose any output.

The child process is not killed if the timeout expires, so in order to cleanup properly a well-behaved application should kill the child process and finish communication:

```
proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()
```

Nota: The data read is buffered in memory, so do not use this method if the data size is large or unlimited.

Alterado na versão 3.3: *timeout* foi adicionado.

Popen.**send_signal**(*signal*)
Envia o sinal *signal* para o filho.

Nota: On Windows, SIGTERM is an alias for *terminate()*. CTRL_C_EVENT and CTRL_BREAK_EVENT can be sent to processes started with a *creationflags* parameter which includes *CREATE_NEW_PROCESS_GROUP*.

Popen.**terminate**()
Stop the child. On POSIX OSs the method sends SIGTERM to the child. On Windows the Win32 API function `TerminateProcess()` is called to stop the child.

Popen.**kill**()
Kills the child. On POSIX OSs the function sends SIGKILL to the child. On Windows *kill()* is an alias for *terminate()*.

Os seguintes atributos também estão disponíveis:

Popen.**args**
O argumento *args* conforme foi passado para *Popen* - uma sequência de argumentos do programa ou então uma única string.

Novo na versão 3.3.

Popen.**stdin**
If the *stdin* argument was *PIPE*, this attribute is a writeable stream object as returned by *open()*. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdin* argument was not *PIPE*, this attribute is `None`.

Popen.**stdout**
If the *stdout* argument was *PIPE*, this attribute is a readable stream object as returned by *open()*. Reading from the stream provides output from the child process. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdout* argument was not *PIPE*, this attribute is `None`.

Popen.**stderr**
If the *stderr* argument was *PIPE*, this attribute is a readable stream object as returned by *open()*. Reading from the stream provides error output from the child process. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stderr* argument was not *PIPE*, this attribute is `None`.

Aviso: Use *communicate()* rather than *.stdin.write*, *.stdout.read* or *.stderr.read* to avoid deadlocks due to any of the other OS pipe buffers filling up and blocking the child process.

Popen.**pid**
O ID de processo do processo filho.

Observe que se você definir o argumento *shell* como `True`, este é o ID do processo do shell gerado.

Popen.**returncode**
The child return code, set by *poll()* and *wait()* (and indirectly by *communicate()*). A `None` value indicates that the process hasn't terminated yet.

Um valor negativo `-N` indica que o filho foi terminado pelo sinal `N` (POSIX apenas).

17.5.4 Windows Popen Helpers

The *STARTUPINFO* class and following constants are only available on Windows.

class subprocess.**STARTUPINFO**(*, *dwFlags=0*, *hStdInput=None*, *hStdOutput=None*, *hStdError=None*, *wShowWindow=0*, *lpAttributeList=None*)
Partial support of the Windows STARTUPINFO structure is used for *Popen* creation. The following attributes can be set by passing them as keyword-only arguments.

Alterado na versão 3.7: Keyword-only argument support was added.

dwFlags
A bit field that determines whether certain *STARTUPINFO* attributes are used when the process creates a window.

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_USESHOWWINDOW
```

hStdInput
If *dwFlags* specifies *STARTF_USESTDHANDLES*, this attribute is the standard input handle for the process. If *STARTF_USESTDHANDLES* is not specified, the default for standard input is the keyboard buffer.

hStdOutput
If *dwFlags* specifies *STARTF_USESTDHANDLES*, this attribute is the standard output handle for the process. Otherwise, this attribute is ignored and the default for standard output is the console window's buffer.

hStdError
If *dwFlags* specifies *STARTF_USESTDHANDLES*, this attribute is the standard error handle for the process. Otherwise, this attribute is ignored and the default for standard error is the console window's buffer.

wShowWindow
If *dwFlags* specifies *STARTF_USESHOWWINDOW*, this attribute can be any of the values that can be specified in the `nCmdShow` parameter for the ShowWindow function, except for `SW_SHOWDEFAULT`. Otherwise, this attribute is ignored.

SW_HIDE is provided for this attribute. It is used when *Popen* is called with `shell=True`.

lpAttributeList
A dictionary of additional attributes for process creation as given in `STARTUPINFOEX`, see UpdateProcThreadAttribute.

Atributos suportados:

handle_list Sequence of handles that will be inherited. *close_fds* must be true if non-empty.

The handles must be temporarily made inheritable by *os.set_handle_inheritable()* when passed to the *Popen* constructor, else *OSError* will be raised with Windows error ERROR_INVALID_PARAMETER (87).

> **Aviso:** In a multithreaded process, use caution to avoid leaking handles that are marked inheritable when combining this feature with concurrent calls to other process creation functions that inherit all handles such as *os.system()*. This also applies to standard handle redirection, which temporarily creates inheritable handles.

Novo na versão 3.7.

Constantes do Windows

The *subprocess* module exposes the following constants.

subprocess.**STD_INPUT_HANDLE**
The standard input device. Initially, this is the console input buffer, CONIN$.

subprocess.**STD_OUTPUT_HANDLE**
The standard output device. Initially, this is the active console screen buffer, CONOUT$.

subprocess.**STD_ERROR_HANDLE**
The standard error device. Initially, this is the active console screen buffer, CONOUT$.

subprocess.**SW_HIDE**
Oculta a janela. Outra janela será ativada.

subprocess.**STARTF_USESTDHANDLES**
Specifies that the *STARTUPINFO.hStdInput*, *STARTUPINFO.hStdOutput*, and *STARTUPINFO.hStdError* attributes contain additional information.

subprocess.**STARTF_USESHOWWINDOW**
Specifies that the *STARTUPINFO.wShowWindow* attribute contains additional information.

subprocess.**CREATE_NEW_CONSOLE**
The new process has a new console, instead of inheriting its parent's console (the default).

subprocess.**CREATE_NEW_PROCESS_GROUP**
A *Popen* `creationflags` parameter to specify that a new process group will be created. This flag is necessary for using *os.kill()* on the subprocess.

This flag is ignored if *CREATE_NEW_CONSOLE* is specified.

subprocess.**ABOVE_NORMAL_PRIORITY_CLASS**
A *Popen* `creationflags` parameter to specify that a new process will have an above average priority.

Novo na versão 3.7.

subprocess.**BELOW_NORMAL_PRIORITY_CLASS**
A *Popen* `creationflags` parameter to specify that a new process will have a below average priority.

Novo na versão 3.7.

subprocess.**HIGH_PRIORITY_CLASS**
A *Popen* `creationflags` parameter to specify that a new process will have a high priority.

Novo na versão 3.7.

subprocess.**IDLE_PRIORITY_CLASS**
A *Popen* `creationflags` parameter to specify that a new process will have an idle (lowest) priority.

Novo na versão 3.7.

subprocess.**NORMAL_PRIORITY_CLASS**
A *Popen* `creationflags` parameter to specify that a new process will have an normal priority. (default)

Novo na versão 3.7.

subprocess.**REALTIME_PRIORITY_CLASS**
A *Popen* `creationflags` parameter to specify that a new process will have realtime priority. You should almost never use REALTIME_PRIORITY_CLASS, because this interrupts system threads that manage mouse input, keyboard input, and background disk flushing. This class can be appropriate for applications that "talk" directly to hardware or that perform brief tasks that should have limited interruptions.

Novo na versão 3.7.

subprocess.**CREATE_NO_WINDOW**
A *Popen* `creationflags` parameter to specify that a new process will not create a window.

Novo na versão 3.7.

subprocess.**DETACHED_PROCESS**
A *Popen* `creationflags` parameter to specify that a new process will not inherit its parent's console. This value cannot be used with CREATE_NEW_CONSOLE.

Novo na versão 3.7.

subprocess.**CREATE_DEFAULT_ERROR_MODE**
A *Popen* `creationflags` parameter to specify that a new process does not inherit the error mode of the calling process. Instead, the new process gets the default error mode. This feature is particularly useful for multithreaded shell applications that run with hard errors disabled.

Novo na versão 3.7.

subprocess.**CREATE_BREAKAWAY_FROM_JOB**
A *Popen* `creationflags` parameter to specify that a new process is not associated with the job.

Novo na versão 3.7.

17.5.5 API de alto nível mais antiga

Antes do Python 3.5, essas três funções constituíam a API de alto nível para subprocesso. Agora você pode usar *run()* em muitos casos, mas muitos códigos existentes chamam essas funções.

subprocess.**call**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)
Run the command described by *args*. Wait for command to complete, then return the *returncode* attribute.

Code needing to capture stdout or stderr should use *run()* instead:

run(…).returncode

To suppress stdout or stderr, supply a value of *DEVNULL*.

The arguments shown above are merely some common ones. The full function signature is the same as that of the *Popen* constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

Nota: Do not use `stdout=PIPE` or `stderr=PIPE` with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

Alterado na versão 3.3: *timeout* foi adicionado.

Alterado na versão 3.7.17: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

`subprocess.`**`check_call`**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)

Run command with arguments. Wait for command to complete. If the return code was zero then return, otherwise raise *CalledProcessError*. The *CalledProcessError* object will have the return code in the *returncode* attribute.

Code needing to capture stdout or stderr should use *run()* instead:

```
run(..., check=True)
```

To suppress stdout or stderr, supply a value of *DEVNULL*.

The arguments shown above are merely some common ones. The full function signature is the same as that of the *Popen* constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

Nota: Do not use `stdout=PIPE` or `stderr=PIPE` with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

Alterado na versão 3.3: *timeout* foi adicionado.

Alterado na versão 3.7.17: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

`subprocess.`**`check_output`**(*args*, *, *stdin=None*, *stderr=None*, *shell=False*, *cwd=None*, *encoding=None*, *errors=None*, *universal_newlines=None*, *timeout=None*, *text=None*, ***other_popen_kwargs*)

Executa o comando com argumentos e retorna sua saída.

If the return code was non-zero it raises a *CalledProcessError*. The *CalledProcessError* object will have the return code in the *returncode* attribute and any output in the *output* attribute.

This is equivalent to:

```
run(..., check=True, stdout=PIPE).stdout
```

The arguments shown above are merely some common ones. The full function signature is largely the same as that of *run()* - most arguments are passed directly through to that interface. However, explicitly passing `input=None` to inherit the parent's standard input file handle is not supported.

By default, this function will return the data as encoded bytes. The actual encoding of the output data may depend on the command being invoked, so the decoding to text will often need to be handled at the application level.

This behaviour may be overridden by setting *text*, *encoding*, *errors*, or *universal_newlines* to `True` as described in *Argumentos Usados Freqüentemente* and *run()*.

To also capture standard error in the result, use `stderr=subprocess.STDOUT`:

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

Novo na versão 3.1.

Alterado na versão 3.3: *timeout* foi adicionado.

Alterado na versão 3.4: Support for the *input* keyword argument was added.

Alterado na versão 3.6: *encoding* and *errors* were added. See *run()* for details.

Novo na versão 3.7: *text* foi adicionado como um atalho mais legível para *universal_newlines*.

Alterado na versão 3.7.17: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

17.5.6 Replacing Older Functions with the `subprocess` Module

In this section, "a becomes b" means that b can be used as a replacement for a.

Nota: All "a" functions in this section fail (more or less) silently if the executed program cannot be found; the "b" replacements raise *OSError* instead.

In addition, the replacements using *check_output()* will fail with a *CalledProcessError* if the requested operation produces a non-zero return code. The output is still available as the *output* attribute of the raised exception.

In the following examples, we assume that the relevant functions have already been imported from the *subprocess* module.

Replacing /bin/sh shell backquote

```
output=`mycmd myarg`
```

torna-se:

```
output = check_output(["mycmd", "myarg"])
```

Replacing shell pipeline

```
output=`dmesg | grep hda`
```

torna-se:

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

The p1.stdout.close() call after starting the p2 is important in order for p1 to receive a SIGPIPE if p2 exits before p1.

Alternatively, for trusted input, the shell's own pipeline support may still be used directly:

```
output=`dmesg | grep hda`
```

torna-se:

```
output=check_output("dmesg | grep hda", shell=True)
```

Substituindo `os.system()`

```
sts = os.system("mycmd" + " myarg")
# becomes
sts = call("mycmd" + " myarg", shell=True)
```

Notas:

- Calling the program through the shell is usually not required.

Um exemplo mais realista ficaria assim:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode, file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

Replacing the `os.spawn` family

Exemplo P_NOWAIT:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

Exemplo P_WAIT:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

Exemplo de vetor:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

Exemplo de ambiente:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

Replacing `os.popen()`, `os.popen2()`, `os.popen3()`

```
(child_stdin, child_stdout) = os.popen2(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```
(child_stdin,
 child_stdout,
 child_stderr) = os.popen3(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin,
 child_stdout,
 child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```
(child_stdin, child_stdout_and_stderr) = os.popen4(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

Return code handling translates as follows:

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
==>
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```

Replacing functions from the `popen2` module

Nota: If the cmd argument to popen2 functions is a string, the command is executed through /bin/sh. If it is a list, the command is directly executed.

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, mode)
==>
p = Popen("somestring", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

```
(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize, mode)
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
```

(continua na próxima página)

(continuação da página anterior)

```
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

`popen2.Popen3` and `popen2.Popen4` basically work as *`subprocess.Popen`*, except that:

- *`Popen`* raises an exception if the execution fails.
- The *capturestderr* argument is replaced with the *stderr* argument.
- `stdin=PIPE` and `stdout=PIPE` must be specified.
- popen2 closes all file descriptors by default, but you have to specify `close_fds=True` with *`Popen`* to guarantee this behavior on all platforms or past Python versions.

17.5.7 Legacy Shell Invocation Functions

This module also provides the following legacy functions from the 2.x `commands` module. These operations implicitly invoke the system shell and none of the guarantees described above regarding security and exception handling consistency are valid for these functions.

`subprocess.`**`getstatusoutput`**`(`*`cmd`*`)`
Return `(exitcode, output)` of executing *cmd* in a shell.

Execute the string *cmd* in a shell with `Popen.check_output()` and return a 2-tuple `(exitcode, output)`. The locale encoding is used; see the notes on *Argumentos Usados Freqüentemente* for more details.

A trailing newline is stripped from the output. The exit code for the command can be interpreted as the return code of subprocess. Example:

```
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(1, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(127, 'sh: /bin/junk: not found')
>>> subprocess.getstatusoutput('/bin/kill $$')
(-15, '')
```

Disponibilidade: POSIX e Windows.

Alterado na versão 3.3.4: Suporte ao Windows foi adicionado.

The function now returns (exitcode, output) instead of (status, output) as it did in Python 3.3.3 and earlier. exitcode has the same value as *`returncode`*.

`subprocess.`**`getoutput`**`(`*`cmd`*`)`
Return output (stdout and stderr) of executing *cmd* in a shell.

Like *`getstatusoutput()`*, except the exit code is ignored and the return value is a string containing the command's output. Example:

```
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

Disponibilidade: POSIX e Windows.

Alterado na versão 3.3.4: Suporte para Windows adicionado.

17.5.8 Notas

Converter uma sequência de argumentos em uma string no Windows

On Windows, an *args* sequence is converted to a string that can be parsed using the following rules (which correspond to the rules used by the MS C runtime):

1. Arguments are delimited by white space, which is either a space or a tab.
2. A string surrounded by double quotation marks is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
3. A double quotation mark preceded by a backslash is interpreted as a literal double quotation mark.
4. Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
5. If backslashes immediately precede a double quotation mark, every pair of backslashes is interpreted as a literal backslash. If the number of backslashes is odd, the last backslash escapes the next double quotation mark as described in rule 3.

Ver também:

shlex Module which provides function to parse and escape command lines.

17.6 `sched` — Event scheduler

Código Fonte: Lib/sched.py

The *sched* module defines a class which implements a general purpose event scheduler:

class sched.**scheduler**(*timefunc=time.monotonic*, *delayfunc=time.sleep*)

The *scheduler* class defines a generic interface to scheduling events. It needs two functions to actually deal with the "outside world" — *timefunc* should be callable without arguments, and return a number (the "time", in any units whatsoever). The *delayfunc* function should be callable with one argument, compatible with the output of *timefunc*, and should delay that many time units. *delayfunc* will also be called with the argument `0` after each event is run to allow other threads an opportunity to run in multi-threaded applications.

Alterado na versão 3.3: *timefunc* and *delayfunc* parameters are optional.

Alterado na versão 3.3: *scheduler* class can be safely used in multi-threaded environments.

Exemplo:

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(a='default'):
...     print("From print_time", time.time(), a)
...
>>> def print_some_times():
...     print(time.time())
...     s.enter(10, 1, print_time)
...     s.enter(5, 2, print_time, argument=('positional',))
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
...     s.run()
...     print(time.time())
...
>>> print_some_times()
```

(continua na próxima página)

(continuação da página anterior)

```
930343690.257
From print_time 930343695.274 positional
From print_time 930343695.275 keyword
From print_time 930343700.273 default
930343700.276
```

17.6.1 Objetos Scheduler

scheduler instances have the following methods and attributes:

scheduler.**enterabs**(*time*, *priority*, *action*, *argument=()*, *kwargs={}*)
Schedule a new event. The *time* argument should be a numeric type compatible with the return value of the *timefunc* function passed to the constructor. Events scheduled for the same *time* will be executed in the order of their *priority*. A lower number represents a higher priority.

Executing the event means executing `action(*argument, **kwargs)`. *argument* is a sequence holding the positional arguments for *action*. *kwargs* is a dictionary holding the keyword arguments for *action*.

Return value is an event which may be used for later cancellation of the event (see *cancel()*).

Alterado na versão 3.3: *argument* parameter is optional.

Novo na versão 3.3: o parâmetro *kwargs* foi adicionado.

scheduler.**enter**(*delay*, *priority*, *action*, *argument=()*, *kwargs={}*)
Schedule an event for *delay* more time units. Other than the relative time, the other arguments, the effect and the return value are the same as those for *enterabs()*.

Alterado na versão 3.3: *argument* parameter is optional.

Novo na versão 3.3: o parâmetro *kwargs* foi adicionado.

scheduler.**cancel**(*event*)
Remove the event from the queue. If *event* is not an event currently in the queue, this method will raise a *ValueError*.

scheduler.**empty**()
Return `True` if the event queue is empty.

scheduler.**run**(*blocking=True*)
Run all scheduled events. This method will wait (using the `delayfunc()` function passed to the constructor) for the next event, then execute it and so on until there are no more scheduled events.

If *blocking* is false executes the scheduled events due to expire soonest (if any) and then return the deadline of the next scheduled call in the scheduler (if any).

Either *action* or *delayfunc* can raise an exception. In either case, the scheduler will maintain a consistent state and propagate the exception. If an exception is raised by *action*, the event will not be attempted in future calls to *run()*.

If a sequence of events takes longer to run than the time available before the next event, the scheduler will simply fall behind. No events will be dropped; the calling code is responsible for canceling events which are no longer pertinent.

Novo na versão 3.3: *blocking* parameter was added.

scheduler.**queue**
Read-only attribute returning a list of upcoming events in the order they will be run. Each event is shown as a *named tuple* with the following fields: time, priority, action, argument, kwargs.

17.7 queue — A synchronized queue class

Código Fonte: Lib/queue.py

The *queue* module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The *Queue* class in this module implements all the required locking semantics. It depends on the availability of thread support in Python; see the *threading* module.

The module implements three types of queue, which differ only in the order in which the entries are retrieved. In a FIFO queue, the first tasks added are the first retrieved. In a LIFO queue, the most recently added entry is the first retrieved (operating like a stack). With a priority queue, the entries are kept sorted (using the *heapq* module) and the lowest valued entry is retrieved first.

Internally, those three types of queues use locks to temporarily block competing threads; however, they are not designed to handle reentrancy within a thread.

In addition, the module implements a "simple" FIFO queue type, *SimpleQueue*, whose specific implementation provides additional guarantees in exchange for the smaller functionality.

The *queue* module defines the following classes and exceptions:

class queue.**Queue**(*maxsize=0*)
Constructor for a FIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

class queue.**LifoQueue**(*maxsize=0*)
Constructor for a LIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

class queue.**PriorityQueue**(*maxsize=0*)
Constructor for a priority queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

The lowest valued entries are retrieved first (the lowest valued entry is the one returned by `sorted(list(entries))[0]`). A typical pattern for entries is a tuple in the form: `(priority_number, data)`.

If the *data* elements are not comparable, the data can be wrapped in a class that ignores the data item and only compares the priority number:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

class queue.**SimpleQueue**
Constructor for an unbounded FIFO queue. Simple queues lack advanced functionality such as task tracking.

Novo na versão 3.7.

exception queue.**Empty**
Exception raised when non-blocking *get()* (or *get_nowait()*) is called on a *Queue* object which is empty.

exception queue.**Full**
Exception raised when non-blocking *put()* (or *put_nowait()*) is called on a *Queue* object which is full.

17.7.1 Objetos Queue

Queue objects (*Queue*, *LifoQueue*, or *PriorityQueue*) provide the public methods described below.

Queue.**qsize**()
Return the approximate size of the queue. Note, qsize() > 0 doesn't guarantee that a subsequent get() will not block, nor will qsize() < maxsize guarantee that put() will not block.

Queue.**empty**()
Return `True` if the queue is empty, `False` otherwise. If empty() returns `True` it doesn't guarantee that a subsequent call to put() will not block. Similarly, if empty() returns `False` it doesn't guarantee that a subsequent call to get() will not block.

Queue.**full**()
Return `True` if the queue is full, `False` otherwise. If full() returns `True` it doesn't guarantee that a subsequent call to get() will not block. Similarly, if full() returns `False` it doesn't guarantee that a subsequent call to put() will not block.

Queue.**put**(*item*, *block=True*, *timeout=None*)
Put *item* into the queue. If optional args *block* is true and *timeout* is `None` (the default), block if necessary until a free slot is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *Full* exception if no free slot was available within that time. Otherwise (*block* is false), put an item on the queue if a free slot is immediately available, else raise the *Full* exception (*timeout* is ignored in that case).

Queue.**put_nowait**(*item*)
Equivalent to `put(item, False)`.

Queue.**get**(*block=True*, *timeout=None*)
Remove and return an item from the queue. If optional args *block* is true and *timeout* is `None` (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *Empty* exception if no item was available within that time. Otherwise (*block* is false), return an item if one is immediately available, else raise the *Empty* exception (*timeout* is ignored in that case).

Prior to 3.0 on POSIX systems, and for all versions on Windows, if *block* is true and *timeout* is `None`, this operation goes into an uninterruptible wait on an underlying lock. This means that no exceptions can occur, and in particular a SIGINT will not trigger a *KeyboardInterrupt*.

Queue.**get_nowait**()
Equivalente a `get(False)`.

Two methods are offered to support tracking whether enqueued tasks have been fully processed by daemon consumer threads.

Queue.**task_done**()
Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each *get()* used to fetch a task, a subsequent call to *task_done()* tells the queue that the processing on the task is complete.

If a *join()* is currently blocking, it will resume when all items have been processed (meaning that a *task_done()* call was received for every item that had been *put()* into the queue).

Raises a *ValueError* if called more times than there were items placed in the queue.

Queue.**join**()
Blocks until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls *task_done()* to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, *join()* unblocks.

Example of how to wait for enqueued tasks to be completed:

```
def worker():
    while True:
        item = q.get()
        if item is None:
            break
        do_work(item)
        q.task_done()

q = queue.Queue()
threads = []
for i in range(num_worker_threads):
    t = threading.Thread(target=worker)
    t.start()
    threads.append(t)

for item in source():
    q.put(item)

# block until all tasks are done
q.join()

# stop workers
for i in range(num_worker_threads):
    q.put(None)
for t in threads:
    t.join()
```

17.7.2 Objetos SimpleQueue

SimpleQueue objects provide the public methods described below.

SimpleQueue.**qsize**()
Return the approximate size of the queue. Note, qsize() > 0 doesn't guarantee that a subsequent get() will not block.

SimpleQueue.**empty**()
Return `True` if the queue is empty, `False` otherwise. If empty() returns `False` it doesn't guarantee that a subsequent call to get() will not block.

SimpleQueue.**put**(*item*, *block=True*, *timeout=None*)
Put *item* into the queue. The method never blocks and always succeeds (except for potential low-level errors such as failure to allocate memory). The optional args *block* and *timeout* are ignored and only provided for compatibility with *Queue.put()*.

CPython implementation detail: This method has a C implementation which is reentrant. That is, a `put()` or `get()` call can be interrupted by another `put()` call in the same thread without deadlocking or corrupting internal state inside the queue. This makes it appropriate for use in destructors such as `__del__` methods or *weakref* callbacks.

SimpleQueue.**put_nowait**(*item*)
Equivalent to `put(item)`, provided for compatibility with *Queue.put_nowait()*.

`SimpleQueue.`**`get`**`(`*block=True*, *timeout=None*`)`
Remove and return an item from the queue. If optional args *block* is true and *timeout* is `None` (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *Empty* exception if no item was available within that time. Otherwise (*block* is false), return an item if one is immediately available, else raise the *Empty* exception (*timeout* is ignored in that case).

`SimpleQueue.`**`get_nowait`**`()`
Equivalente a `get(False)`.

Ver também:

Class ***multiprocessing.Queue*** A queue class for use in a multi-processing (rather than multi-threading) context.

collections.deque is an alternative implementation of unbounded queues with fast atomic *append()* and *popleft()* operations that do not require locking.

A seguir, os módulos de suporte para alguns dos serviços acima:

17.8 :mod:thread— API de segmentação de baixo nível

Este módulo fornece primitivos de baixo nível para trabalhar com vários tópicos (também chamados: dfn: *light-weight processes* ou: dfn:‘ tasks‘) — vários tópicos de controle compartilhando seu espaço de dados global. Para sincronização, bloqueios simples (também chamados de: dfn: *mutexes* ou: dfn:‘ binary semaphores‘) são fornecidos. O módulo: mod: *threading* fornece uma API de segmentação mais fácil de usar e de nível mais alto, construída sobre este módulo.

Alterado na versão 3.7: Este módulo costumava ser opcional, agora está sempre disponível.

Este módulo define as seguintes constantes e funções:

exception `_thread.`**`error`**
Gerado em erros específicos de segmento.

Alterado na versão 3.3: Este é agora um sinônimo do built-in: exc: *RuntimeError*.

`_thread.`**`LockType`**
Este é o tipo de objetos de bloqueio.

`_thread.`**`start_new_thread`**`(`*function*, *args*`[`, *kwargs* `]`)
Inicie um novo encadeamento e retorne seu identificador. O encadeamento executa a função * function * com a lista de argumentos * args * (que deve ser uma tupla). O argumento opcional * kwargs * especifica um dicionário de argumentos de palavras-chave. Quando a função retorna, o segmento sai silenciosamente. Quando a função termina com uma exceção não tratada, um rastreamento de pilha é impresso e, em seguida, o segmento sai (mas outros segmentos continuam a ser executados).

`_thread.`**`interrupt_main`**`()`
Simula o efeito de *signal.SIGINT* signal chegando na thread principal. Uma thread pode usar essa função para interromper a thread principal.

Se *signal.SIGINT* não for manipulado pelo Python (foi definido como *signal.SIG_DFL* ou *signal.SIG_IGN*), essa função não faz nada.

`_thread.`**`exit`**`()`
Gera a exceção:exc:*SystemExit*. Quando não for detectada, o thread sairá silenciosamente.

`_thread.`**`allocate_lock`**`()`
Retorna um novo objeto de bloqueio. Métodos de bloqueio são descritos abaixo. O bloqueio é desativado inicialmente.

_thread.**get_ident**()
Retorna o 'identificador de thread' do thread atual. Este é um número inteiro diferente de zero. Seu valor não tem significado direto; pretende-se que seja um cookie mágico para ser usado, por exemplo, para indexar um dicionário de dados específicos do thread. identificadores de thread podem ser reciclados quando um thread sai e outro é criado.

_thread.**stack_size**([*size*])
Retorna o tamanho da pilha de threads usado ao criar novos threads. O argumento opcional *size* especifica o tamanho da pilha a ser usado para threads criados posteriormente e deve ser 0 (usar plataforma ou padrão configurado) ou um valor inteiro positivo de pelo menos 32.768 (32 KiB). Se *size* não for especificado, 0 será usado. Se a alteração do tamanho da pilha de threads não for suportada, um :exc: *RuntimeError* será gerado. Se o tamanho da pilha especificado for inválido, um :exc: *ValueError* será aumentado e o tamanho da pilha não será modificado. Atualmente, 0 KiB é o valor mínimo de tamanho de pilha suportado para garantir espaço suficiente para o próprio intérprete. Observe que algumas plataformas podem ter restrições específicas sobre valores para o tamanho da pilha, como exigir um tamanho mínimo de pilha> 32 KiB ou exigir alocação em múltiplos do tamanho da página de memória do sistema - a documentação da plataforma deve ser consultada para obter mais informações (4 páginas KiB são comuns; usar múltiplos de 4096 para o tamanho da pilha é a abordagem sugerida na ausência de informações mais específicas).

Availability: Windows, systems with POSIX threads.

_thread.**TIMEOUT_MAX**
O valor máximo permitido para o parâmetro * timeout * de :meth: *Lock.acquire*. A especificação de um tempo limite maior que esse valor gerará um: exc: *OverflowError*.

Novo na versão 3.2.

Os objetos de bloqueio têm os seguintes métodos:

lock.**acquire**(*waitflag=1*, *timeout=-1*)
Sem nenhum argumento opcional, esse método adquire o bloqueio incondicionalmente, se necessário, aguardando até que seja liberado por outro encadeamento (apenas um encadeamento por vez pode adquirir um bloqueio - esse é o motivo da sua existência).

Se o argumento inteiro *waitflag* estiver presente, a ação dependerá do seu valor: se for zero, o bloqueio será adquirido apenas se puder ser adquirido imediatamente sem aguardar, enquanto se for diferente de zero, o bloqueio será adquirido incondicionalmente, conforme acima.

Se o argumento de ponto flutuante *timeout* estiver presente e positivo, ele especificará o tempo máximo de espera em segundos antes de retornar. Um argumento negativo *timeout* especifica uma espera ilimitada. Você não pode especificar um *timeout* se *waitflag* for zero.

O valor de retorno é `True` se o bloqueio for adquirido com sucesso, se não `False`.

Alterado na versão 3.2: O parâmetro *timeout* é novo.

Alterado na versão 3.2: As aquisições de bloqueio agora podem ser interrompidas por sinais no POSIX.

lock.**release**()
Libera o bloqueio. O bloqueio deve ter sido adquirido anteriormente, mas não necessariamente pela mesma thread.

lock.**locked**()
Retorna o status do bloqueio: `True` se tiver sido adquirido por alguma thread, `False` se não for o caso.

Além desses métodos, os objetos de bloqueio também podem ser usados através da instrução `with`, por exemplo:

```
import _thread

a_lock = _thread.allocate_lock()
```

(continua na próxima página)

(continuação da página anterior)

```
with a_lock:
    print("a_lock is locked while this executes")
```

Ressalvas:

- Threads interagem estranhamente com interrupções: a exceção *KeyboardInterrupt* será recebida por uma thread arbitrário. (Quando o módulo *signal* está disponível, as interrupções sempre vão para a thread principal.)
- Chamar *sys.exit()* ou levantar a exceção *SystemExit* é o equivalente a chamar *_thread.exit()*.
- Não é possível interromper o método `acquire()` em um bloqueio — a exceção *KeyboardInterrupt* ocorrerá após o bloqueio ter sido adquirido.
- Quando a thread principal se encerra, é definido pelo sistema se as outras threads sobrevivem. Na maioria dos sistemas, elas são eliminadas sem executar cláusulas `try` ... `finally` ou executar destruidores de objetos.
- Quando a thread principal é encerrada, ela não realiza nenhuma limpeza usual (exceto que as cláusulas `try` ... `finally` são honradas) e os arquivos de E/S padrão não são liberados.

17.9 _dummy_thread — Substituição direta para o módulo _thread

Código-fonte: Lib/_dummy_thread.py

Obsoleto desde a versão 3.7: O Python agora sempre tem a segmentação ativada. Por favor, use *_thread* (ou. melhor, *threading*).

Este módulo fornece uma interface duplicada para o módulo *_thread*. A ideia era ele ser importado quando o módulo *_thread* não fosse fornecido em uma plataforma.

Tenha cuidado para não usar este módulo onde o deadlock pode ocorrer a partir de uma segmento que está sendo criado, bloqueando a espera pela criação de outro segmento. Isso geralmente ocorre com o bloqueio de E/S.

17.10 dummy_threading — Substituição drop-in para o módulo threading

Código-fonte: Lib/dummy_threading.py

Obsoleto desde a versão 3.7: O Python agora sempre tem a segmentação ativada. Por favor use *threading*.

Este módulo fornece uma interface duplicada para o módulo *threading*. A ideia é que ele fosse importado quando o módulo *_thread* não fosse fornecido em uma plataforma.

Tenha cuidado para não usar este módulo onde o deadlock pode ocorrer a partir de uma segmento que está sendo criado, bloqueando a espera pela criação de outro segmento. Isso geralmente ocorre com o bloqueio de E/S.

CAPÍTULO 18

contextvars — Variáveis de contexto

Este módulo fornece APIs para gerenciar, armazenar e acessar o estado local do contexto. A classe *ContextVar* é usada para declarar e trabalhar com *Variáveis de Contexto*. A função *copy_context()* e a classe *Context* devem ser usadas para gerenciar o contexto atual em frameworks assíncronos.

Os gerenciadores de contexto que possuem estado devem usar Variáveis de Contexto ao invés de *threading.local()* para evitar que seu estado vaze para outro código inesperadamente, quando usado em código concorrente.

Veja também a **PEP 567** para detalhes adicionais.

Novo na versão 3.7.

18.1 Variáveis de contexto

class contextvars.**ContextVar**(*name*[, *, *default*])

Esta classe é usada para declarar uma nova variável de contexto, como, por exemplo:

```
var: ContextVar[int] = ContextVar('var', default=42)
```

O parâmetro obrigatório *name* é usado para fins de introspecção e depuração.

O parâmetro somente-nomeado opcional *default* é retornado por *ContextVar.get()* quando nenhum valor para a variável é encontrado no contexto atual.

Importante: Variáveis de Contexto devem ser criadas no nível do módulo superior e nunca em fechamentos. Os objetos *Context* contêm referências fortes a variáveis de contexto que evitam que as variáveis de contexto sejam coletadas como lixo corretamente.

name

O nome da variável. Esta é uma propriedade somente leitura.

Novo na versão 3.7.1.

get([*default*])
Retorna um valor para a variável de contexto para o contexto atual.

Se não houver valor para a variável no contexto atual, o método vai:

- retornar o valor do argumento *default* do método, se fornecido; ou
- retornar o valor padrão para a variável de contexto, se ela foi criada com uma; ou
- levantar uma *LookupError*.

set(*value*)
Chame para definir um novo valor para a variável de contexto no contexto atual.

O argumento *value* obrigatório é o novo valor para a variável de contexto.

Retorna um objeto *Token* que pode ser usado para restaurar a variável ao seu valor anterior através do método *ContextVar.reset()*.

reset(*token*)
Redefine a variável de contexto para o valor que tinha antes de :meth: *ContextVar.set.* que criou o *token*, ser usado.

Por exemplo:

```
var = ContextVar('var')

token = var.set('new value')
# code that uses 'var'; var.get() returns 'new value'.
var.reset(token)

# After the reset call the var has no value again, so
# var.get() would raise a LookupError.
```

class contextvars.**Token**
Objetos *token* são retornados pelo método *ContextVar.set()*. Eles podem ser passados para o método *ContextVar.reset()* para reverter o valor da variável para o que era antes do *set* correspondente.

Token.**var**
Uma propriedade somente leitura. Aponta para o objeto *ContextVar* que criou o token.

Token.**old_value**
Uma propriedade somente leitura. Defina com o valor que a variável tinha antes da chamada do método *ContextVar.set()* que criou o token. Ele aponta para `Token.MISSING` é a variável não foi definida antes da chamada.

Token.**MISSING**
Um objeto marcador usado por `Token.old_value`.

18.2 Gerenciamento de Contexto Manual

contextvars.**copy_context**()
Retorna uma cópia do objeto *Context* atual.

O trecho a seguir obtém uma cópia do contexto atual e imprime todas as variáveis e seus valores que são definidos nele:

```
ctx: Context = copy_context()
print(list(ctx.items()))
```

A função tem uma complexidade O(1), ou seja, funciona igualmente rápido para contextos com algumas variáveis de contexto e para contextos que têm muitas delas.

class contextvars.**Context**

Um mapeamento de *ContextVars* para seus valores.

`Context()` cria um contexto vazio sem valores nele. Para obter uma cópia do contexto atual, use a função *copy_context()*.

Context implementa a interface *collections.abc.Mapping*.

run(*callable*, **args*, ***kwargs*)

Executa o código `callable(*args, **kwargs)` no objeto contexto em que o método *run* é chamado. Retorna o resultado da execução ou propaga uma exceção, se ocorrer.

Quaisquer mudanças em quaisquer variáveis de contexto que *callable* faça estarão contidas no objeto de contexto:

```
var = ContextVar('var')
var.set('spam')

def main():
    # 'var' was set to 'spam' before
    # calling 'copy_context()' and 'ctx.run(main)', so:
    # var.get() == ctx[var] == 'spam'

    var.set('ham')

    # Now, after setting 'var' to 'ham':
    # var.get() == ctx[var] == 'ham'

ctx = copy_context()

# Any changes that the 'main' function makes to 'var'
# will be contained in 'ctx'.
ctx.run(main)

# The 'main()' function was run in the 'ctx' context,
# so changes to 'var' are contained in it:
# ctx[var] == 'ham'

# However, outside of 'ctx', 'var' is still set to 'spam':
# var.get() == 'spam'
```

O método levanta uma *RuntimeError* quando chamado no mesmo objeto de contexto de mais de uma thread do sistema operacional, ou quando chamado recursivamente.

copy()

Retorna uma cópia rasa do objeto contexto.

var in context

Retorna `True` se *context* tem uma variável para *var* definida; do contrário, retorna `False`.

context[var]

Retorna o valor da variável *ContextVar* *var*. Se a variável não for definida no objeto contexto, uma *KeyError* é levantada.

get(*var*[, *default*])

Retorna o valor para *var* se *var* tiver o valor no objeto contexto. Caso contrário, retorna *default*. Se *default* não for fornecido, retorna `None`.

iter(context)
Retorna um iterador sobre as variáveis armazenadas no objeto contexto.

len(proxy)
Retorna o número das variáveis definidas no objeto contexto.

keys()
Retorna uma lista de todas as variáveis no objeto contexto.

values()
Retorna uma lista dos valores de todas as variáveis no objeto contexto.

items()
Retorna uma lista de tuplas de 2 elementos contendo todas as variáveis e seus valores no objeto contexto.

18.3 Suporte a asyncio

Variáveis de contexto encontram suporte nativo em *asyncio* e estão prontas para serem usadas sem qualquer configuração extra. Por exemplo, aqui está um servidor simples de eco, que usa uma variável de contexto para disponibilizar o endereço de um cliente remoto na Task que lida com esse cliente:

```
import asyncio
import contextvars

client_addr_var = contextvars.ContextVar('client_addr')

def render_goodbye():
    # The address of the currently handled client can be accessed
    # without passing it explicitly to this function.

    client_addr = client_addr_var.get()
    return f'Good bye, client @ {client_addr}\n'.encode()

async def handle_request(reader, writer):
    addr = writer.transport.get_extra_info('socket').getpeername()
    client_addr_var.set(addr)

    # In any code that we call is now possible to get
    # client's address by calling 'client_addr_var.get()'.

    while True:
        line = await reader.readline()
        print(line)
        if not line.strip():
            break
        writer.write(line)

    writer.write(render_goodbye())
    writer.close()

async def main():
    srv = await asyncio.start_server(
        handle_request, '127.0.0.1', 8081)

    async with srv:
        await srv.serve_forever()
```

(continua na próxima página)

(continuação da página anterior)

```
asyncio.run(main())

# To test it you can use telnet:
#     telnet 127.0.0.1 8081
```

CAPÍTULO 19

Comunicação em Rede e Interprocesso

Os módulos descritos neste capítulo fornecem mecanismos para a comunicação em rede e entre processos.

Alguns módulos funcionam apenas para dois processos que estão na mesma máquina como, por exemplo, *signal* e *mmap*. Outros módulos possuem suporte a protocolos de rede que dois ou mais processos podem usar para se comunicar entre máquinas.

A lista de módulos descritos neste capítulo é:

19.1 asyncio — E/S assíncrona

Olá Mundo!

```
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

# Python 3.7+
asyncio.run(main())
```

asyncio é uma biblioteca para escrever código **simultâneo** usando a sintaxe **async/await**.

O asyncio é usado como uma base para várias estruturas assíncronas do Python que fornecem rede e servidores web de alto desempenho, bibliotecas de conexão de banco de dados, filas de tarefas distribuídas etc.

asyncio geralmente serve perfeitamente para código de rede **estruturado** de alto nível e vinculado a E/S.

asyncio fornece um conjunto de APIs de **alto nível** para:

- *executar corrotinas do Python* simultaneamente e ter controle total sobre sua execução;
- realizar *IPC e E/S de rede*;
- controlar *subprocessos*;
- distribuir tarefas por meio de *filas*;
- *sincronizar* código simultâneo;

Além disso, há APIs de **baixo nível** para *desenvolvedores de biblioteca e framework* para:

- criar e gerenciar *loops de eventos*, que fornecem APIs assíncronas para `networking`, executando `subprocesses`, lidando com `OS signals`, etc;
- implementar protocolos eficientes usando *transportes*;
- *fazer uma ponte* sobre bibliotecas baseadas em chamadas e codificar com a sintaxe de async/await.

Referência

19.1.1 Corrotinas e Tarefas

Esta seção descreve APIs assíncronas de alto nível para trabalhar com corrotinas e tarefas.

Coroutines

Coroutines declared with async/await syntax is the preferred way of writing asyncio applications. For example, the following snippet of code (requires Python 3.7+) prints "hello", waits 1 second, and then prints "world":

```
>>> import asyncio

>>> async def main():
...     print('hello')
...     await asyncio.sleep(1)
...     print('world')

>>> asyncio.run(main())
hello
world
```

Perceba que simplesmente chamar uma corrotina não irá agendá-la para ser executada:

```
>>> main()
<coroutine object main at 0x1053bb7c8>
```

Para realmente executar uma corrotina, asyncio fornece três mecanismos principais:

- A função: func:*asyncio.run* para executar a função "main()" do ponto de entrada no nível mais alto (veja o exemplo acima.)
- Aguardando uma corrotina. O seguinte trecho de código exibirá "hello" após esperar por 1 segundo e, em seguida, exibirá "world" após esperar por *outros* 2 segundos:

  ```
  import asyncio
  import time

  async def say_after(delay, what):
      await asyncio.sleep(delay)
      print(what)

  async def main():
      print(f"started at {time.strftime('%X')}")

      await say_after(1, 'hello')
      await say_after(2, 'world')

      print(f"finished at {time.strftime('%X')}")

  asyncio.run(main())
  ```

  Resultado esperado:

  ```
  started at 17:13:52
  hello
  world
  finished at 17:13:55
  ```

- A função *asyncio.create_task()* para executar corrotinas concorrentemente como *Tasks* asyncio.

  Vamo modificar o exemplo acima e executar duas corrotinas `say_after` *concorrentemente*:

```
async def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
        say_after(2, 'world'))

    print(f"started at {time.strftime('%X')}")

    # Wait until both tasks are completed (should take
    # around 2 seconds.)
    await task1
    await task2

    print(f"finished at {time.strftime('%X')}")
```

Perceba que a saída esperada agora mostra que o trercho de código é executado 1 segundo mais rápido do que antes:

```
started at 17:14:32
hello
world
finished at 17:14:34
```

Aguardáveis

Dizemos que um objeto é um objeto **aguardável** se ele pode ser usado em uma expressão `await`. Muitas APIs asyncio são projetadas para aceitar aguardáveis.

Existem três tipos principais de objetos *aguardáveis*: **corrotinas**, **Tarefas**, e **Futuros**.

Coroutines

Corrotinas Python são *aguardáveis* e portanto podem ser aguardadas a partir de outras corrotinas:

```
import asyncio

async def nested():
    return 42

async def main():
    # Nothing happens if we just call "nested()".
    # A coroutine object is created but not awaited,
    # so it *won't run at all*.
    nested()

    # Let's do it differently now and await it:
    print(await nested())  # will print "42".

asyncio.run(main())
```

Importante: Nesta documentação, o termo "corrotina" pode ser usado para dois conceitos intimamente relacionados:

- uma *função de corrotina*: uma função `async def`;

- um *objeto de corrotina*: um objeto retornado ao chamar uma *função de corrotina*.

asyncio também suporta corrotinas legadas *baseadas em geradores*.

Tarefas

Tarefas são usadas para agendar corrotinas *concorrentemente*.

Quando uma corrotina é envolta em uma *tarefa* com funções como `asyncio.create_task()`, a corrotina é automaticamente agendada para executar em breve:

```
import asyncio

async def nested():
    return 42

async def main():
    # Schedule nested() to run soon concurrently
    # with "main()".
    task = asyncio.create_task(nested())

    # "task" can now be used to cancel "nested()", or
    # can simply be awaited to wait until it is complete:
    await task

asyncio.run(main())
```

Futuros

Um `Future` é um objeto aguardável especial de **baixo nível** que representa um **resultado eventual** de uma operação assíncrona.

Quando um objeto Future é *aguardado* isso significa que a corrotina irá esperar até que o Future seja resolvido em algum outro local.

Objetos Future em asyncio são necessários para permitir que código baseado em função de retorno seja utilizado com async/await.

Normalmente **não existe necessidade** em criar objetos Future no nível de código da aplicação.

Objetos Future, algumas vezes expostos por bibliotecas e algumas APIs asyncio, podem ser aguardados:

```
async def main():
    await function_that_returns_a_future_object()

    # this is also valid:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

Um bom exemplo de uma função de baixo nível que retorna um objeto Future é `loop.run_in_executor()`.

Executando um programa asyncio

asyncio.**run**(*coro*, *, *debug=False*)

Executa a *corrotina* *coro* e retorna o resultado.

This function runs the passed coroutine, taking care of managing the asyncio event loop and *finalizing asynchronous generators*.

Esta função não pode ser chamada quando outro ciclo de eventos asyncio está executando na mesma thread.

Se *debug* for `True`, o ciclo de eventos irá ser executado em modo debug.

Esta função sempre cria um novo ciclo de eventos e fecha-o no final. Ela deve ser usada como um ponto de entrada principal para programas asyncio, e deve idealmente ser chamada apenas uma vez.

Exemplo:

```
async def main():
    await asyncio.sleep(1)
    print('hello')

asyncio.run(main())
```

Novo na versão 3.7: **Important:** this function has been added to asyncio in Python 3.7 on a *provisional basis*.

Criando Tarefas

asyncio.**create_task**(*coro*)

Envolva a *corrotina* *coro* em uma *Task* e agende sua execução. Retorne o objeto Task.

A tarefa é executada no loop e retornada por *get_running_loop()*, *RuntimeError* é levantado se não existir nenhum loop na thread atual.

Esta função foi **adicionada no Python 3.7**. Antes do Python 3.7, a função de baixo nível *asyncio.ensure_future()* pode ser usada ao invés:

```
async def coro():
    ...

# In Python 3.7+
task = asyncio.create_task(coro())
...

# This works in all Python versions but is less readable
task = asyncio.ensure_future(coro())
...
```

Novo na versão 3.7.

Dormindo

coroutine asyncio.**sleep**(*delay*, *result=None*, *, *loop=None*)
Bloqueia por *delay* segundos.

Se *result* é fornecido, é retornado para o autor da chamada quando a corrotina termina.

`sleep()` sempre suspende a tarefa atual, permitindo que outras tarefas sejam executadas.

The *loop* argument is deprecated and scheduled for removal in Python 3.10.

Exemplo de uma corrotina exibindo a data atual a cada segundo durante 5 segundos:

```
import asyncio
import datetime

async def display_date():
    loop = asyncio.get_running_loop()
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

asyncio.run(display_date())
```

Executando tarefas concorrentemente

awaitable asyncio.**gather**(**aws*, *loop=None*, *return_exceptions=False*)
Executa *objetos aguardáveis* na sequência *aws* de forma *concorrente*.

Se qualquer aguardável em *aws* é uma corrotina, ele é automaticamente agendado como uma Tarefa.

Se todos os aguardáveis forem concluídos com sucesso, o resultado é uma lista agregada de valores retornados. A ordem dos valores resultantes corresponde a ordem dos aguardáveis em *aws*.

Se *return_exceptions* for `False` (valor padrão), a primeira exceção levantada é imediatamente propagada para a tarefa que espera em `gather()`. Outros aguardáveis na sequência *aws* **não serão cancelados** e irão continuar a executar.

Se *return_exceptions* for `True`, exceções são tratadas da mesma forma que que resultados com sucesso, e agregadas na lista de resultados.

Se `gather()` for *cancelado*, todos os aguardáveis que foram submetidos (que não foram concluídos ainda) também são *cancelados*.

Se qualquer Task ou Future da sequência *aws* for *cancelado*, ele é tratado como se tivesse levantado *CancelledError* – a chamada para `gather()` **não** é cancelada neste caso. Isso existe para prevenir que o cancelamento de uma Task/Future submetida ocasione outras Tasks/Futures a serem cancelados.

Exemplo:

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({i})...")
```

(continua na próxima página)

(continuação da página anterior)

```
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")

async def main():
    # Schedule three calls *concurrently*:
    await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )

asyncio.run(main())

# Expected output:
#
#     Task A: Compute factorial(2)...
#     Task B: Compute factorial(2)...
#     Task C: Compute factorial(2)...
#     Task A: factorial(2) = 2
#     Task B: Compute factorial(3)...
#     Task C: Compute factorial(3)...
#     Task B: factorial(3) = 6
#     Task C: Compute factorial(4)...
#     Task C: factorial(4) = 24
```

Alterado na versão 3.7: Se *gather* por si mesmo for cancelado, o cancelamento é propagado independente de *return_exceptions*.

Protegendo contra cancelamento

awaitable asyncio.**shield**(*aw*, *, *loop=None*)

Protege um *objeto aguardável* de ser `cancelado`.

Se *aw* é uma corrotina, ela é automaticamente agendada como uma Task.

A instrução:

```
res = await shield(something())
```

is equivalent to:

```
res = await something()
```

exceto que se a corrotina contendo-a for cancelada, a Task executando em `something()` não é cancelada. Do ponto de vista de `something()`, o cancelamento não aconteceu. Apesar do autor da chamada ainda estar cancelado, então a expressão "await" ainda levanta um `CancelledError`.

Se `something()` é cancelada por outros meios (isto é, dentro ou a partir de si mesma) isso também iria cancelar `shield()`.

Se for desejado ignorar completamente os cancelamentos (não recomendado) a função `shield()` deve ser combinada com uma cláusula try/except, conforme abaixo:

```
try:
    res = await shield(something())
```

(continua na próxima página)

(continuação da página anterior)

```
except CancelledError:
    res = None
```

Tempo limite

coroutine asyncio.**wait_for**(*aw*, *timeout*, *, *loop=None*)

Espera o *aguardável* *aw* concluir sem ultrapassar o tempo limite "timeout".

Se *aw* é uma corrotina, ela é automaticamente agendada como uma Task.

timeout pode ser `None`, ou um ponto flutuante, ou um número inteiro de segundos para aguardar. Se *timeout* é `None`, aguarda até o future encerrar.

Se o tempo limite timeout for atingido, ele cancela a tarefa e levanta *asyncio.TimeoutError*.

Para evitar o *cancelamento* da tarefa, envolva-a com *shield()*.

The function will wait until the future is actually cancelled, so the total wait time may exceed the *timeout*.

Se ele for cancelado, o future *aw* também é cancelado.

The *loop* argument is deprecated and scheduled for removal in Python 3.10.

Exemplo:

```
async def eternity():
    # Sleep for one hour
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # Wait for at most 1 second
    try:
        await asyncio.wait_for(eternity(), timeout=1.0)
    except asyncio.TimeoutError:
        print('timeout!')

asyncio.run(main())

# Expected output:
#
#     timeout!
```

Alterado na versão 3.7: Quando *aw* é cancelado devido a um tempo limite, `wait_for` aguarda que *aw* seja cancelado. Anteriormente, ele levantava *asyncio.TimeoutError* imediatamente.

Primitivas de Espera

coroutine asyncio.**wait**(*aws*, *, *loop=None*, *timeout=None*, *return_when=ALL_COMPLETED*)

Executa *objetos aguardáveis* no *aws* definidos concorrentemente e bloqueia até atingir a condição especificada por *return_when*.

Se qualquer aguardável em *aws* for uma corrotina, ela é automaticamente agendada como uma tarefa. Passar objetos que são corrotinas para `wait()` diretamente está descontinuado, pois leva a *comportamentos confusos*.

Retorna dois conjuntos de Tarefas/Futuros: `(done, pending)`.

Utilização:

```
done, pending = await asyncio.wait(aws)
```

The *loop* argument is deprecated and scheduled for removal in Python 3.10.

timeout (um ponto flutuante ou inteiro), se especificado, pode ser usado para controlar o número máximo de segundos para aguardar antes de retornar.

Perceba que esta função não levanta *asyncio.TimeoutError*. Futuros ou Tarefas que não estão concluídas quando o tempo limite é excedido são simplesmente retornadas no segundo conjunto.

return_when indica quando esta função deve retornar. Ele deve ser uma das seguintes constantes:

Constante	Description (descrição)
FIRST_COMPLETED	A função irá retornar quando qualquer futuro terminar ou for cancelado.
FIRST_EXCEPTION	A função irá retornar quando qualquer futuro encerrar levantando uma exceção. Se nenhum futuro levantar uma exceção, então é equivalente a ALL_COMPLETED.
ALL_COMPLETED	A função irá retornar quando todos os futuros encerrarem ou forem cancelados.

Diferente de *wait_for()*, wait() não cancela os futuros quando um tempo limite é atingido.

Nota: wait() agenda corrotinas como Tarefas automaticamente e posteriormente retorna esses objetos Tasks criados implicitamente em conjuntos (done, pending). Portanto o seguinte código não irá funcionar como esperado:

```
async def foo():
    return 42

coro = foo()
done, pending = await asyncio.wait({coro})

if coro in done:
    # This branch will never be run!
```

Aqui está a forma como o trecho de código acima pode ser consertado:

```
async def foo():
    return 42

task = asyncio.create_task(foo())
done, pending = await asyncio.wait({task})

if task in done:
    # Everything will work as expected now.
```

Passar objetos corrotina para wait() diretamente foi descontinuado.

---

asyncio.**as_completed**(*aws*, *, *loop=None*, *timeout=None*)

Run *awaitable objects* in the *aws* set concurrently. Return an iterator of *Future* objects. Each Future object returned represents the earliest result from the set of the remaining awaitables.

Levanta *asyncio.TimeoutError* se o tempo limite ocorrer antes que todos os futuros tenham encerrado.

Exemplo:

```
for f in as_completed(aws):
    earliest_result = await f
    # ...
```

Agendando a partir de outras Threads

asyncio.**run_coroutine_threadsafe**(*coro*, *loop*)

Envia uma corrotina para o ciclo de eventos fornecido. Seguro para thread.

Retorna um *concurrent.futures.Future* para aguardar pelo resultado de outra thread do sistema operacional.

Esta função destina-se a ser chamada partir de uma thread diferente do sistema operacional, da qual o ciclo de eventos está executando. Exemplo:

```
# Create a coroutine
coro = asyncio.sleep(1, result=3)

# Submit the coroutine to a given loop
future = asyncio.run_coroutine_threadsafe(coro, loop)

# Wait for the result with an optional timeout argument
assert future.result(timeout) == 3
```

Se uma exceção for levantada na corrotina, o Future retornado será notificado. Isso também pode ser usado para cancelar a tarefa no ciclo de eventos:

```
try:
    result = future.result(timeout)
except asyncio.TimeoutError:
    print('The coroutine took too long, cancelling the task...')
    future.cancel()
except Exception as exc:
    print(f'The coroutine raised an exception: {exc!r}')
else:
    print(f'The coroutine returned: {result!r}')
```

Veja a seção *concorrência e multithreading* da documentação.

Ao contrário de outras funções asyncio, esta função requer que o argumento *loop* seja passado explicitamente.

Novo na versão 3.5.1.

Instrospecção

asyncio.**current_task**(*loop=None*)

Retorna a instância *Task* atualmente em execução, ou `None` se nenhuma tarefa estiver executando.

Se *loop* for `None`, então *get_running_loop()* é usado para obter o laço atual.

Novo na versão 3.7.

asyncio.**all_tasks**(*loop=None*)

Retorna um conjunto de objetos *Task* ainda não concluídos a serem executados pelo laço.

Se *loop* for `None`, então *get_running_loop()* é usado para obter o laço atual.

Novo na versão 3.7.

Objeto Task

class asyncio.**Task**(*coro*, *, *loop=None*)

Um objeto *similar a Futuro* que executa uma *corrotina* Python. Não é seguro para thread.

Tarefas são usadas para executar corrotinas em ciclo de eventos. Se uma corrotina espera por um Futuro, a Tarefa suspende a execução da corrotina e aguarda a conclusão do Futuro. Quando o futuro é *concluído*, a execução da corrotina contida é retomada.

Ciclo de eventos usam agendamento cooperativo: um ciclo de evento executa uma Tarefa de cada vez. Enquanto uma Tarefa aguarda de um Futuro, o ciclo de eventos executa outras Tarefas, funções de retorno, ou executa operações de IO.

Use a função de alto nível *asyncio.create_task()* para criar Tarefas, ou as funções de baixo nível *loop.create_task()* ou *ensure_future()*. Instanciação manual de Tarefas é desencorajado.

Para cancelar uma Tarefa em execução, use o método *cancel()*. Chamar ele fará com que a Tarefa levante uma exceção *CancelledError* dentro da corrotina contida. Se a corrotina estiver esperando por um objeto Future durante o cancelamento, o objeto Future será cancelado.

cancelled() pode ser usado para verificar se a Tarefa foi cancelada. O método retorna `True` se a corrotina envolta não suprimiu a exceção *CancelledError* e foi na verdade cancelada.

asyncio.Task herda de *Future* todas as suas APIs exceto *Future.set_result()* e *Future.set_exception()*.

Tarefas suportam o módulo *contextvars*. Quando a Tarefa é criada, ela copia o contexto atual e posteriormente executa sua corrotina no contexto copiado.

Alterado na versão 3.7: Adicionado suporte para o módulo *contextvars*.

cancel()

Solicita o cancelamento da Tarefa.

Isto prepara para uma exceção *CancelledError* ser lançada na corrotina contida no próximo ciclo do ciclo de eventos.

A corrotina então tem uma chance de limpar ou até mesmo negar a requisição, suprimindo a exceção com um bloco `try` ... ... `except CancelledError` ... `finally`. Portanto, ao contrário de *Future.cancel()*, *Task.cancel()* não garante que a Tarefa será cancelada, apesar que suprimir o cancelamento completamente não é comum, e é ativamente desencorajado.

O seguinte exemplo ilustra como corrotinas podem interceptar o cancelamento de requisições:

```
async def cancel_me():
    print('cancel_me(): before sleep')

    try:
        # Wait for 1 hour
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        print('cancel_me(): cancel sleep')
        raise
    finally:
        print('cancel_me(): after sleep')

async def main():
    # Create a "cancel_me" Task
    task = asyncio.create_task(cancel_me())

    # Wait for 1 second
```

(continua na próxima página)

(continuação da página anterior)

```
    await asyncio.sleep(1)

    task.cancel()
    try:
        await task
    except asyncio.CancelledError:
        print("main(): cancel_me is cancelled now")

asyncio.run(main())

# Expected output:
#
#     cancel_me(): before sleep
#     cancel_me(): cancel sleep
#     cancel_me(): after sleep
#     main(): cancel_me is cancelled now
```

cancelled()
Retorna `True` se a Tarefa for *cancelada*.

A Tarefa é *cancelada* quando o cancelamento foi requisitado com *cancel()* e a corrotina contida propagou a exceção *CancelledError* gerada nela.

done()
Retorna `True` se a Tarefa estiver *concluída*.

Uma Tarefa está *concluída* quando a corrotina contida retornou um valor, ou levantou uma exceção, ou a Tarefa foi cancelada.

result()
Retorna o resultado da Tarefa.

Se a Tarefa estiver *concluída*, o resultado da corrotina contida é retornado (ou se a corrotina levantou uma exceção, essa exceção é re-levantada.)

Se a Tarefa foi *cancelada*, este método levanta uma exceção *CancelledError*.

Se o resultado da Tarefa não estiver disponível ainda, este método levanta uma exceção *InvalidStateError*.

exception()
Retorna a exceção de uma Tarefa.

Se a corrotina contida levantou uma exceção, essa exceção é retornada. Se a corrotina contida retornou normalmente, este método retorna `None`.

Se a Tarefa foi *cancelada*, este método levanta uma exceção *CancelledError*.

Se a Tarefa não estiver *concluída* ainda, este método levanta uma exceção *InvalidStateError*.

add_done_callback(*callback*, *, *context=None*)
Adiciona uma função de retorno para ser executada quando a Tarefa estiver *concluída*.

Este método deve ser usado apenas em código de baixo nível baseado em funções de retorno.

Veja a documentação para *Future.add_done_callback()* para mais detalhes.

remove_done_callback(*callback*)
Remove *callback* da lista de funções de retorno.

Este método deve ser usado apenas em código de baixo nível baseado em funções de retorno.

Veja a documentação do método *Future.remove_done_callback()* para mais detalhes.

get_stack(*, *limit=None*)
Retorna a lista de frames da pilha para esta Tarefa.

Se a corrotina contida não estiver concluída, isto retorna a pilha onde ela foi suspensa. Se a corrotina foi concluída com sucesso ou foi cancelada, isto retorna uma lista vazia. Se a corrotina foi terminada por uma exceção, isto retorna a lista de frames do traceback (situação da pilha de execução).

Os quadros são sempre ordenados dos mais antigos para os mais recentes.

Apenas um frame da pilha é retornado para uma corrotina suspensa.

O argumento opcional *limit* define o o número de frames máximo para retornar; por padrão todos os frames disponíveis são retornados. O ordenamento da lista retornada é diferente dependendo se uma pilha ou um traceback (situação da pilha de execução) é retornado: os frames mais recentes de uma pilha são retornados, mas os frames mais antigos de um traceback são retornados. (Isso combina com o comportamento do módulo traceback.)

print_stack(*, *limit=None*, *file=None*)
Exibe a pilha ou situação da pilha de execução para esta Tarefa.

Isto produz uma saída similar a do módulo traceback para frames recuperados por *get_stack()*.

O argumento *limit* é passado para *get_stack()* diretamente.

O argumento *file* é um fluxo de entrada e saída para o qual a saída é escrita; por padrão a saída é escrita para *sys.stderr*.

classmethod all_tasks(*loop=None*)
Return a set of all tasks for an event loop.

By default all tasks for the current event loop are returned. If *loop* is `None`, the *get_event_loop()* function is used to get the current loop.

This method is **deprecated** and will be removed in Python 3.9. Use the *asyncio.all_tasks()* function instead.

classmethod current_task(*loop=None*)
Return the currently running task or `None`.

If *loop* is `None`, the *get_event_loop()* function is used to get the current loop.

This method is **deprecated** and will be removed in Python 3.9. Use the *asyncio.current_task()* function instead.

Corrotinas baseadas em gerador

Nota: Suporte para corrotinas baseadas em gerador está **descontinuado** e agendado para ser removido no Python 3.10.

Corrotinas baseadas em gerador antecedem a sintaxe async/await. Elas são geradores Python que usam expressões `yield from` para aguardar Futuros e outras corrotinas.

Corrotinas baseadas em gerador devem ser decoradas com *@asyncio.coroutine*, apesar disso não ser forçado.

`@asyncio.`**coroutine**
Decorador para marcar corrotinas baseadas em gerador.

Este decorador permite que corrotinas legadas baseadas em gerador sejam compatíveis com código async/await:

```
@asyncio.coroutine
def old_style_coroutine():
    yield from asyncio.sleep(1)

async def main():
    await old_style_coroutine()
```

This decorator is **deprecated** and is scheduled for removal in Python 3.10.

Este decorador não deve ser usado para corrotinas `async def`.

`asyncio.`**`iscoroutine`**(*obj*)
Retorna `True` se *obj* é um *objeto corrotina*.

Este método é diferente de *`inspect.iscoroutine()`* porque ele retorna `True` para corrotinas baseadas em gerador.

`asyncio.`**`iscoroutinefunction`**(*func*)
Retorna `True` se *func* é uma *função de corrotina*.

Este método é diferente de *`inspect.iscoroutinefunction()`* porque ele retorna `True` para funções de corrotina baseadas em gerador, decoradas com *`@coroutine`*.

19.1.2 Fluxos

Streams são conexões de rede de alto-nível assincronias/espera-pronta. Streams permitem envios e recebimentos de dados sem usar retornos de chamadas ou protocolos de baixo nível.

Aqui está um exemplo de um cliente TCP realizando eco, escrito usando streams asyncio:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

Veja também a seção *Exemplos* abaixo.

Funções Stream

As seguintes funções asyncio de alto nível podems ser usadas para criar e trabalhar com fluxos:

coroutine asyncio.**open_connection**(*host=None*, *port=None*, ***, *loop=None*, *limit=None*, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*)

Estabelece uma conexão de rede e retorna um par de objetos `(reader, writer)`.

Os objetos *reader* e *writer* retornados são instâncias das classes `StreamReader` e `StreamWriter`.

O argumento *loop* é opcional e sempre pode ser determinado automaticamente, quando esta função é esperada a partir de uma corrotina.

limit determina o tamanho limite do buffer usado pela instância `StreamReader` retornada. Por padrão, *limit* é definido em 64 KiB.

O resto dos argumentos são passados diretamente para `loop.create_connection()`.

Novo na versão 3.7: O parâmetro *ssl_handshake_timeout*.

coroutine asyncio.**start_server**(*client_connected_cb*, *host=None*, *port=None*, ***, *loop=None*, *limit=None*, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *ssl_handshake_timeout=None*, *start_serving=True*)

Inicia um soquete no servidor.

A função de retorno *client_connected_cb* é chamada sempre que uma nova conexão de um cliente é estabelecida. Ela recebe um par `(reader, writer)` como dois argumentos, instâncias das classes `StreamReader` e `StreamWriter`.

client_connected_cb pode ser simplesmente algo chamável ou uma *função de corrotina*; se ele for uma função de corrotina, ele será automaticamente agendado como uma `Task`.

O argumento *loop* é opcional e pode sempre ser determinado automaticamente quando este método é esperado a partir de uma corrotina.

limit determina o tamanho limite do buffer usado pela instância `StreamReader` retornada. Por padrão, *limit* é definido em 64 KiB.

O resto dos argumentos são passados diretamente para `loop.create_server()`.

Novo na versão 3.7: Os parâmetros *ssl_handshake_timeout* e *start_serving*.

Soquetes Unix

coroutine asyncio.**open_unix_connection**(*path=None*, ***, *loop=None*, *limit=None*, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*)

Estabelece uma conexão de soquete Unix e retorna um par com `(reader, writer)`.

Similar a `open_connection()`, mas opera em soquetes Unix.

Veja também a documentação do método `loop.create_unix_connection()`.

Disponibilidade: Unix.

Novo na versão 3.7: O parâmetro *ssl_handshake_timeout*.

Alterado na versão 3.7: O parâmetro *path* agora pode ser um *objeto caminho ou similar*

coroutine asyncio.**start_unix_server**(*client_connected_cb*, *path=None*, *, *loop=None*, *limit=None*, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *start_serving=True*)

Inicia um servidor com soquete Unix.

Similar a `start_server()`, mas funciona com soquetes Unix.

Veja também a documentação do método `loop.create_unix_server()`.

Disponibilidade: Unix.

Novo na versão 3.7: Os parâmetros *ssl_handshake_timeout* e *start_serving*.

Alterado na versão 3.7: O parâmetro *path* agora pode ser um *objeto caminho ou similar*.

StreamReader

class asyncio.**StreamReader**

Representa um objeto leitor, que fornece APIs para ler dados a partir do stream de entrada/saída.

Não é recomendado instanciar objetos *StreamReader* diretamente; use `open_connection()` e `start_server()` ao invés disso.

coroutine read(*n=-1*)

Executa a leitura de até *n* bytes. Se *n* não for informado, ou for definido para `-1`, executa a leitura até que EOF (fim do arquivo) seja atingido, e retorna todos os bytes lidos.

Se EOF foi recebido e o buffer interno estiver vazio, retorna um objeto `bytes` vazio.

coroutine readline()

Lê uma linha, onde "line" é uma sequência de bytes encerrando com `\n`.

Se EOF é recebido e `\n` não foi encontrado, o método retorna os dados parcialmente lidos.

Se EOF for recebido e o buffer interno estiver vazio, retorna um objeto `bytes` vazio.

coroutine readexactly(*n*)

Lê exatamente *n* bytes.

Levanta um `IncompleteReadError` se EOF é atingido antes que *n* sejam lidos. Use o atributo `IncompleteReadError.partial` para obter os dados parcialmente lidos.

coroutine readuntil(*separator=b'\n'*)

Lê dados a partir do stream até que *separator* seja encontrado.

Ao ter sucesso, os dados e o separador serão removidos do buffer interno (consumido). Dados retornados irão incluir o separador no final.

Se a quantidade de dados lidos excede o limite configurado para o stream, uma exceção `LimitOverrunError` é levantada, e os dados são deixados no buffer interno e podem ser lidos novamente.

Se EOF for atingido antes que o separador completo seja encontrado, uma exceção `IncompleteReadError` é levantada, e o buffer interno é resetado. O atributo `IncompleteReadError.partial` pode conter uma parte do separador.

Novo na versão 3.5.2.

at_eof()

Retorna `True` se o buffer estiver vazio e `feed_eof()` foi chamado.

StreamWriter

class asyncio.**StreamWriter**

Representa um objeto de escrita que fornece APIs para escrever dados para o stream de IO.

Não é recomendado instanciar objetos *StreamWriter* diretamente; use *open_connection()* e *start_server()* ao invés.

can_write_eof()

Retorna `True` se o transporte subjacente suporta o método *write_eof()*, `False` caso contrário.

write_eof()

Fecha o extremo de escrita do stream após os dados no buffer de escrita terem sido descarregados.

transport

Retorna o transporte asyncio subjacente.

get_extra_info(*name*, *default=None*)

Acessa informações de transporte opcionais; veja *BaseTransport.get_extra_info()* para detalhes.

write(*data*)

Write *data* to the stream.

This method is not subject to flow control. Calls to `write()` should be followed by *drain()*.

writelines(*data*)

Write a list (or any iterable) of bytes to the stream.

This method is not subject to flow control. Calls to `writelines()` should be followed by *drain()*.

coroutine drain()

Aguarda até que seja apropriado continuar escrevendo no stream. Exemplo:

```
writer.write(data)
await writer.drain()
```

Este é um método de controle de fluxo que interage com o buffer de entrada e saída de escrita subjacente. Quando o tamanho do buffer atinge a marca d'agua alta, *drain()* bloqueia até que o tamanho do buffer seja drenado para a marca d'água baixa, e a escrita possa continuar. Quando não existe nada que cause uma espera, o método *drain()* retorna imediatamente.

close()

Close the stream.

is_closing()

Retorna `True` se o stream estiver fechado ou em processo de ser fechado.

Novo na versão 3.7.

coroutine wait_closed()

Aguarda até que o stream seja fechado.

Deve ser chamado após *close()* para aguardar até que a conexão subjacente esteja fechada.

Novo na versão 3.7.

Exemplos

Cliente para eco TCP usando streams

Cliente de eco TCP usando a função *asyncio.open_connection()*:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()

asyncio.run(tcp_echo_client('Hello World!'))
```

Ver também:

O exemplo de *protocolo do cliente para eco TCP* usa o método de baixo nível *loop.create_connection()*.

Servidor eco TCP usando streams

Servidor eco TCP usando a função *asyncio.start_server()*:

```
import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')

    print(f"Received {message!r} from {addr!r}")

    print(f"Send: {message!r}")
    writer.write(data)
    await writer.drain()

    print("Close the connection")
    writer.close()

async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addr = server.sockets[0].getsockname()
    print(f'Serving on {addr}')

    async with server:
        await server.serve_forever()
```

(continua na próxima página)

(continuação da página anterior)

```
asyncio.run(main())
```

Ver também:

O exemplo de *protocolo eco de servidor TCP* utiliza o método `loop.create_server()`.

Obtém headers HTTP

Exemplo simples consultando cabeçalhos HTTP da URL passada na linha de comando:

```
import asyncio
import urllib.parse
import sys

async def print_http_headers(url):
    url = urllib.parse.urlsplit(url)
    if url.scheme == 'https':
        reader, writer = await asyncio.open_connection(
            url.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(
            url.hostname, 80)

    query = (
        f"HEAD {url.path or '/'} HTTP/1.0\r\n"
        f"Host: {url.hostname}\r\n"
        f"\r\n"
    )

    writer.write(query.encode('latin-1'))
    while True:
        line = await reader.readline()
        if not line:
            break

        line = line.decode('latin1').rstrip()
        if line:
            print(f'HTTP header> {line}')

    # Ignore the body, close the socket
    writer.close()

url = sys.argv[1]
asyncio.run(print_http_headers(url))
```

Utilização:

```
python example.py http://example.com/path/page.html
```

ou com HTTPS:

```
python example.py https://example.com/path/page.html
```

Registra um soquete aberto para aguardar por dados usando streams

Corrotina aguardando até que um soquete receba dados usando a função *open_connection()*:

```
import asyncio
import socket

async def wait_for_data():
    # Get a reference to the current event loop because
    # we want to access low-level APIs.
    loop = asyncio.get_running_loop()

    # Create a pair of connected sockets.
    rsock, wsock = socket.socketpair()

    # Register the open socket to wait for data.
    reader, writer = await asyncio.open_connection(sock=rsock)

    # Simulate the reception of data from the network
    loop.call_soon(wsock.send, 'abc'.encode())

    # Wait for data
    data = await reader.read(100)

    # Got data, we are done: close the socket
    print("Received:", data.decode())
    writer.close()

    # Close the second socket
    wsock.close()

asyncio.run(wait_for_data())
```

Ver também:

O exemplo de *registro de um soquete aberto para aguardar por dados usando um protocolo* utiliza um protocolo de baixo nível e o método *loop.create_connection()*.

O exemplo para *monitorar um descritor de arquivo para leitura de eventos* utiliza o método de baixo nível *loop.add_reader()* para monitorar um descritor de arquivo.

19.1.3 Synchronization Primitives

asyncio synchronization primitives are designed to be similar to those of the *threading* module with two important caveats:

- asyncio primitives are not thread-safe, therefore they should not be used for OS thread synchronization (use *threading* for that);
- methods of these synchronization primitives do not accept the *timeout* argument; use the *asyncio.wait_for()* function to perform operations with timeouts.

asyncio has the following basic synchronization primitives:

- *Lock*
- *Event*
- *Condition*

- *Semaphore*
- *BoundedSemaphore*

Lock

class asyncio.**Lock**(*, *loop=None*)

Implements a mutex lock for asyncio tasks. Not thread-safe.

An asyncio lock can be used to guarantee exclusive access to a shared resource.

The preferred way to use a Lock is an `async with` statement:

```
lock = asyncio.Lock()

# ... later
async with lock:
    # access shared state
```

which is equivalent to:

```
lock = asyncio.Lock()

# ... later
await lock.acquire()
try:
    # access shared state
finally:
    lock.release()
```

coroutine acquire()

Acquire the lock.

This method waits until the lock is *unlocked*, sets it to *locked* and returns `True`.

When more than one coroutine is blocked in *acquire()* waiting for the lock to be unlocked, only one coroutine eventually proceeds.

Acquiring a lock is *fair*: the coroutine that proceeds will be the first coroutine that started waiting on the lock.

release()

Release the lock.

When the lock is *locked*, reset it to *unlocked* and return.

If the lock is *unlocked*, a *RuntimeError* is raised.

locked()

Return `True` if the lock is *locked*.

Evento

class asyncio.**Event**(*, *loop=None*)
An event object. Not thread-safe.

An asyncio event can be used to notify multiple asyncio tasks that some event has happened.

An Event object manages an internal flag that can be set to *true* with the *set()* method and reset to *false* with the *clear()* method. The *wait()* method blocks until the flag is set to *true*. The flag is set to *false* initially.

Exemplo:

```
async def waiter(event):
    print('waiting for it ...')
    await event.wait()
    print('... got it!')

async def main():
    # Create an Event object.
    event = asyncio.Event()

    # Spawn a Task to wait until 'event' is set.
    waiter_task = asyncio.create_task(waiter(event))

    # Sleep for 1 second and set the event.
    await asyncio.sleep(1)
    event.set()

    # Wait until the waiter task is finished.
    await waiter_task

asyncio.run(main())
```

coroutine wait()
Wait until the event is set.

If the event is set, return `True` immediately. Otherwise block until another task calls *set()*.

set()
Set the event.

All tasks waiting for event to be set will be immediately awakened.

clear()
Clear (unset) the event.

Tasks awaiting on *wait()* will now block until the *set()* method is called again.

is_set()
Return `True` if the event is set.

Condição

class asyncio.**Condition**(*lock=None*, *, *loop=None*)

A Condition object. Not thread-safe.

An asyncio condition primitive can be used by a task to wait for some event to happen and then get exclusive access to a shared resource.

In essence, a Condition object combines the functionality of an *Event* and a *Lock*. It is possible to have multiple Condition objects share one Lock, which allows coordinating exclusive access to a shared resource between different tasks interested in particular states of that shared resource.

The optional *lock* argument must be a *Lock* object or None. In the latter case a new Lock object is created automatically.

The preferred way to use a Condition is an async with statement:

```
cond = asyncio.Condition()

# ... later
async with cond:
    await cond.wait()
```

which is equivalent to:

```
cond = asyncio.Condition()

# ... later
await cond.acquire()
try:
    await cond.wait()
finally:
    cond.release()
```

coroutine acquire()

Acquire the underlying lock.

This method waits until the underlying lock is *unlocked*, sets it to *locked* and returns True.

notify(*n=1*)

Wake up at most *n* tasks (1 by default) waiting on this condition. The method is no-op if no tasks are waiting.

The lock must be acquired before this method is called and released shortly after. If called with an *unlocked* lock a *RuntimeError* error is raised.

locked()

Return True if the underlying lock is acquired.

notify_all()

Wake up all tasks waiting on this condition.

This method acts like *notify()*, but wakes up all waiting tasks.

The lock must be acquired before this method is called and released shortly after. If called with an *unlocked* lock a *RuntimeError* error is raised.

release()

Release the underlying lock.

When invoked on an unlocked lock, a *RuntimeError* is raised.

coroutine wait()
Wait until notified.

If the calling task has not acquired the lock when this method is called, a *RuntimeError* is raised.

This method releases the underlying lock, and then blocks until it is awakened by a *notify()* or *notify_all()* call. Once awakened, the Condition re-acquires its lock and this method returns True.

coroutine wait_for(*predicate*)
Wait until a predicate becomes *true*.

The predicate must be a callable which result will be interpreted as a boolean value. The final value is the return value.

Semaphore

class asyncio.**Semaphore**(*value=1*, *, *loop=None*)
A Semaphore object. Not thread-safe.

A semaphore manages an internal counter which is decremented by each *acquire()* call and incremented by each *release()* call. The counter can never go below zero; when *acquire()* finds that it is zero, it blocks, waiting until some task calls *release()*.

The optional *value* argument gives the initial value for the internal counter (1 by default). If the given value is less than 0 a *ValueError* is raised.

The preferred way to use a Semaphore is an async with statement:

```
sem = asyncio.Semaphore(10)

# ... later
async with sem:
    # work with shared resource
```

which is equivalent to:

```
sem = asyncio.Semaphore(10)

# ... later
await sem.acquire()
try:
    # work with shared resource
finally:
    sem.release()
```

coroutine acquire()
Acquire a semaphore.

If the internal counter is greater than zero, decrement it by one and return True immediately. If it is zero, wait until a *release()* is called and return True.

locked()
Returns True if semaphore can not be acquired immediately.

release()
Release a semaphore, incrementing the internal counter by one. Can wake up a task waiting to acquire the semaphore.

Unlike *BoundedSemaphore*, *Semaphore* allows making more release() calls than acquire() calls.

BoundedSemaphore

class asyncio.**BoundedSemaphore**(*value=1*, *, *loop=None*)
A bounded semaphore object. Not thread-safe.

Bounded Semaphore is a version of *Semaphore* that raises a *ValueError* in *release()* if it increases the internal counter above the initial *value*.

Obsoleto desde a versão 3.7: Acquiring a lock using `await lock` or `yield from lock` and/or `with` statement (`with await lock`, `with (yield from lock)`) is deprecated. Use `async with lock` instead.

19.1.4 Subprocesso

Esta seção descreve APIs async/await asyncio de alto nível para criar e gerenciar subprocessos.

Aqui está um exemplo de como asyncio pode executar um comando shell e obter o seu resultado:

```
import asyncio

async def run(cmd):
    proc = await asyncio.create_subprocess_shell(
        cmd,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE)

    stdout, stderr = await proc.communicate()

    print(f'[{cmd!r} exited with {proc.returncode}]')
    if stdout:
        print(f'[stdout]\n{stdout.decode()}')
    if stderr:
        print(f'[stderr]\n{stderr.decode()}')

asyncio.run(run('ls /zzz'))
```

irá exibir:

```
['ls /zzz' exited with 1]
[stderr]
ls: /zzz: No such file or directory
```

Devido ao fato que todas as funções de subprocessos asyncio são assíncronas e asyncio fornece muitas ferramentas para trabalhar com tais funções, é fácil executar e monitorar múltiplos subprocessos em paralelo. É na verdade trivial modificar o exemplo acima para executar diversos comandos simultaneamente:

```
async def main():
    await asyncio.gather(
        run('ls /zzz'),
        run('sleep 1; echo "hello"'))

asyncio.run(main())
```

Veja também a subseção *Exemplos*.

Criando Subprocessos

coroutine asyncio.**create_subprocess_exec**(*program*, **args*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*, *limit=None*, ***kwds*)

Cria um subprocesso.

O argumento *limit* define o buffer limite para os wrappers *StreamReader* para `Process.stdout` e `Process.stderr` (se *subprocess.PIPE* for passado para os argumentos *stdout* e *stderr*).

Retorna uma instância de *Process*.

Veja a documentação de *loop.subprocess_exec()* para outros parâmetros.

coroutine asyncio.**create_subprocess_shell**(*cmd*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*, *limit=None*, ***kwds*)

Executa o comando *cmd* no shell.

O argumento *limit* define o buffer limite para os wrappers *StreamReader* para `Process.stdout` e `Process.stderr` (se *subprocess.PIPE* for passado para os argumentos *stdout* e *stderr*).

Retorna uma instância de *Process*.

Veja a documentação de *loop.subprocess_shell()* para outros parâmetros.

Importante: É responsabilidade da aplicação garantir que todos os espaços em branco e caracteres especiais tenham aspas apropriadamente para evitar vulnerabilidades de injeção de shell. A função *shlex.quote()* pode ser usada para escapar espaços em branco e caracteres especiais de shell apropriadamente em strings que serão usadas para construir comandos shell.

Nota: The default asyncio event loop implementation on **Windows** does not support subprocesses. Subprocesses are available for Windows if a *ProactorEventLoop* is used. See *Subprocess Support on Windows* for details.

Ver também:

asyncio também tem as seguintes APIs *de baixo nível* para trabalhar com subprocessos: *loop.subprocess_exec()*, *loop.subprocess_shell()*, *loop.connect_read_pipe()*, *loop.connect_write_pipe()*, assim como os *Transportes de Subprocesso* e *Protocolos de Subprocesso*.

Constantes

asyncio.subprocess.**PIPE**

Pode ser passado para os parâmetros *stdin*, *stdout* ou *stderr*.

Se *PIPE* for passado para o argumento *stdin*, o atributo *Process.stdin* irá apontar para uma instância *StreamWriter*.

Se *PIPE* for passado para os argumentos *stdout* ou *stderr*, os atributos *Process.stdout* e *Process.stderr* irão apontar para instâncias *StreamReader*.

asyncio.subprocess.**STDOUT**

Valor especial que pode ser usado como o argumento *stderr* e indica que a saída de erro padrão deve ser redirecionada para a saída padrão.

asyncio.subprocess.**DEVNULL**

Valor especial que pode ser usado como argumento *stdin*, *stdout* ou *stderr* para funções de criação de processos. Ele indica que o arquivo especial *os.devnull* será usado para o fluxo de subprocesso correspondente.

Interagindo com Subprocesso

Ambas as funções *create_subprocess_exec()* e *create_subprocess_shell()* retornam instâncias da classe *Process*. *Process* é um wrapper de alto nível que permite a comunicação com subprocessos e observar eles serem completados.

class asyncio.subprocess.**Process**

Um objeto que envolve processos do sistema operacional criados pelas funções *create_subprocess_exec()* e *create_subprocess_shell()*.

Esta classe é projetada para ter uma API similar a classe *subprocess.Popen*, mas existem algumas diferenças notáveis:

- ao contrário de Popen, Instâncias de Process não tem um equivalente ao método *poll()*;
- os métodos *communicate()* e *wait()* não tem um parâmetro *timeout*: utilize a função *wait_for()*;
- o método *Process.wait()* é assíncrono, enquanto que o método *subprocess.Popen.wait()* é implementado como um laço bloqueante para indicar que está ocupado;
- o parâmetro *universal_newlines* não é suportado.

Esta classe *não é seguro para thread*.

Veja também a seção *Subprocesso e Threads*.

coroutine wait()

Aguarda o processo filho encerrar.

Define e retorna o atributo *returncode*.

Nota: Este método pode entrar em deadlock ao usar `stdout=PIPE` ou `stderr=PIPE` e o processo filho gera tantas saídas que ele bloqueia a espera pelo encadeamento de buffer do sistema operacional para aceitar mais dados. Use o método *communicate()* ao usar encadeamentos para evitar essa condição.

---

coroutine communicate(*input=None*)

Interage com processo:

1. envia dados para *stdin* (se *input* for diferente de `None`);
2. lê dados a partid de *stdout* e *stderr*, até que EOF (fim do arquivo) seja atingido;
3. aguarda o processo encerrar.

O argumento opcional *input* é a informação (objeto *bytes*) que será enviada para o processo filho.

Retorna uma tupla `(stdout_data, stderr_data)`.

Se qualquer exceção *BrokenPipeError* ou *ConnectionResetError* for levantada ao escrever *input* em *stdin*, a exceção é ignorada. Esta condição ocorre quando o processo encerra antes de todos os dados serem escritos em *stdin*.

Se for desejado enviar dados para o *stdin* do processo, o mesmo precisa ser criado com `stdin=PIPE`. De forma similar, para obter qualquer coisa além de `None` na tupla resultante, o processo precisa ser criado com os argumentos `stdout=PIPE` e/ou `stderr=PIPE`.

Perceba que, os dados lidos são armazenados em um buffer na memória, então não use este método se o tamanho dos dados é grande ou ilimitado.

send_signal(*signal*)

Envia o sinal *signal* para o processo filho.

Nota: No Windows, `SIGTERM` é um apelido para *`terminate()`*. `CTRL_C_EVENT` e `CTRL_BREAK_EVENT` podem ser enviados para processos iniciados com um parâmetro *creationflags*, o qual inclui `CREATE_NEW_PROCESS_GROUP`.

`terminate``()`
Interrompe o processo filho.

Em sistemas POSIX este método envia *`signal.SIGTERM`* para o processo filho.

No Windows a função `TerminateProcess()` da API Win32 é chamada para interromper o processo filho.

`kill``()`
Mata o (processo) filho

Em sistemas POSIX este método envia `SIGKILL` para o processo filho.

No Windows este método é um atalho para *`terminate()`*.

`stdin`
Fluxo de entrada padrão (*`StreamWriter`*) ou `None` se o processo foi criado com `stdin=None`.

`stdout`
Fluxo de saída padrão (*`StreamReader`*) ou `None` se o processo foi criado com `stdout=None`.

`stderr`
Erro de fluxo padrão (*`StreamReader`*) ou `None` se o processo foi criado com `stderr=None`.

Aviso: Use o método *`communicate()`* ao invés de *`process.stdin.write()`*, *`await process.stdout.read()`* ou *`await process.stderr.read`*. Isso evita deadlocks devido a fluxos pausando a leitura ou escrita, e bloqueando o processo filho.

`pid`
Número de identificação do processo (PID).

Perceba que para processos criados pela função *`create_subprocess_shell()`* function, este atributo é o PID do console gerado.

`returncode`
Retorna o código do processo quando o mesmo terminar.

Um valor `None` indica que o processo ainda não terminou.

Um valor negativo `-N` indica que o filho foi terminado pelo sinal `N` (POSIX apenas).

Subprocesso e Threads

Standard asyncio event loop supports running subprocesses from different threads, but there are limitations:

- An event loop must run in the main thread.
- The child watcher must be instantiated in the main thread before executing subprocesses from other threads. Call the *`get_child_watcher()`* function in the main thread to instantiate the child watcher.

Note that alternative event loop implementations might not share the above limitations; please refer to their documentation.

Ver também:

A seção *Concorrência e multithreading em asyncio*.

Exemplos

Um exemplo de uso da classe *Process* para controlar um subprocesso e a classe *StreamReader* para ler a partir da sua saída padrão.

O subprocesso é criado pela função *create_subprocess_exec()*:

```
import asyncio
import sys

async def get_date():
    code = 'import datetime; print(datetime.datetime.now())'

    # Create the subprocess; redirect the standard output
    # into a pipe.
    proc = await asyncio.create_subprocess_exec(
        sys.executable, '-c', code,
        stdout=asyncio.subprocess.PIPE)

    # Read one line of output.
    data = await proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # Wait for the subprocess exit.
    await proc.wait()
    return line

if sys.platform == "win32":
    asyncio.set_event_loop_policy(
        asyncio.WindowsProactorEventLoopPolicy())

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

Veja também o *mesmo exemplo* escrito usando APIs de baixo nível.

19.1.5 Filas

filas asyncio são projetadas para serem similar a classes do módulo *queue*. Apesar que filas asyncio não são seguro para thread, elas são projetadas para serem usadas especificamente em código async/await.

Perceba que métodos de filas asyncio não possuem um parâmetro *timeout*; use a função *asyncio.wait_for()* para realizar operações de fila com um tempo limite timeout.

Veja também a seção *Exemplos* abaixo.

Queue

class asyncio.**Queue**(*maxsize=0*, *, *loop=None*)
Uma fila onde o primeiro a entrar, é o primeiro a sair (FIFO - First In First Out).

Se *maxsize* for menor que ou igual a zero, o tamanho da fila é infinito. Se ele for um inteiro maior que `0`, então `await put()` bloqueia quando a fila atingir *maxsize* até que um item seja removido por *get()*.

Ao contrário da biblioteca padrão de threading *queue*, o tamanho da fila é sempre conhecido e pode ser obtido através da chamada do método *qsize()*.

Esta classe *não é seguro para thread*.

maxsize
Número de itens permitidos na fila.

empty()
Retorna `True` se a fila estiver vazia, `False` caso contrário.

full()
Retorna `True` se existem *maxsize* itens na fila.

Se a fila foi inicializada com `maxsize=0` (o padrão), então *full()* nunca retorna `True`.

coroutine get()
Remove e retorna um item da fila. Se a fila estiver vazia, aguarda até que um item esteja disponível.

get_nowait()
Retorna um item se houver um imediatamente disponível, caso contrário levanta *QueueEmpty*.

coroutine join()
Bloqueia até que todos os itens na fila tenham sido recebidos e processados.

A contagem de tarefas inacabadas aumenta sempre que um item é adicionado a fila. A contagem diminui sempre que uma corrotina consumidora chama *task_done()* para indicar que o item foi recuperado e todo o trabalho nele foi concluído. Quando a contagem de tarefas inacabadas chega a zero, *join()* desbloqueia.

coroutine put(*item*)
Adiciona um item na fila. Se a fila estiver cheia, aguarda até que uma posição livre esteja disponível antes de adicionar o item.

put_nowait(*item*)
Coloca um item na fila sem bloqueá-la.

Se nenhuma posição livre estiver imediatamente disponível, levanta *QueueFull*.

qsize()
Retorna o número de itens na fila.

task_done()
Indica que a tarefa anteriormente enfileira está concluída.

Usada por fila de consumidores. Para cada *get()* usado para buscar uma tarefa, uma chamada subsequente para *task_done()* avisa a fila, que o processamento na tarefa está concluído.

Se um *join()* estiver sendo bloqueado no momento, ele irá continuar quando todos os itens tiverem sido processados (significando que uma chamada *task_done()* foi recebida para cada item que foi chamado o método *put()* para colocar na fila).

Levanta *ValueError* se chamada mais vezes do que a quantidade de itens existentes na fila.

Fila de Prioridade

class asyncio.**PriorityQueue**
Uma variante de *Queue*; recupera entradas em ordem de prioridade (mais baixas primeiro).

Entradas são tipicamente tuplas no formato `(priority_number, data)`.

Filas LIFO (último a entrar, primeiro a sair)

class asyncio.**LifoQueue**
Uma variante de *Queue* que recupera as entradas adicionadas mais recentemente primeiro (último a entrar, primeiro a sair).

Exceções

exception asyncio.**QueueEmpty**
Esta exceção é levantada quando o método *get_nowait()* é chamado em uma fila vazia.

exception asyncio.**QueueFull**
Exceção levantada quando o método *put_nowait()* é chamado em uma fila que atingiu seu *maxsize*.

Exemplos

Filas podem ser usadas para distribuir cargas de trabalho entre diversas tarefas concorrentes:

```
import asyncio
import random
import time


async def worker(name, queue):
    while True:
        # Get a "work item" out of the queue.
        sleep_for = await queue.get()

        # Sleep for the "sleep_for" seconds.
        await asyncio.sleep(sleep_for)

        # Notify the queue that the "work item" has been processed.
        queue.task_done()

        print(f'{name} has slept for {sleep_for:.2f} seconds')


async def main():
    # Create a queue that we will use to store our "workload".
    queue = asyncio.Queue()

    # Generate random timings and put them into the queue.
    total_sleep_time = 0
    for _ in range(20):
        sleep_for = random.uniform(0.05, 1.0)
        total_sleep_time += sleep_for
        queue.put_nowait(sleep_for)
```

(continua na próxima página)

(continuação da página anterior)

```
    # Create three worker tasks to process the queue concurrently.
    tasks = []
    for i in range(3):
        task = asyncio.create_task(worker(f'worker-{i}', queue))
        tasks.append(task)

    # Wait until the queue is fully processed.
    started_at = time.monotonic()
    await queue.join()
    total_slept_for = time.monotonic() - started_at

    # Cancel our worker tasks.
    for task in tasks:
        task.cancel()
    # Wait until all worker tasks are cancelled.
    await asyncio.gather(*tasks, return_exceptions=True)

    print('====')
    print(f'3 workers slept in parallel for {total_slept_for:.2f} seconds')
    print(f'total expected sleep time: {total_sleep_time:.2f} seconds')


asyncio.run(main())
```

19.1.6 Exceções

exception asyncio.**TimeoutError**
A operação excedeu o prazo estabelecido.

Importante: Esta exceção é diferente da exceção embutida *TimeoutError*.

---

exception asyncio.**CancelledError**
A operação foi cancelada.

Esta exceção pode ser capturada para executar operações personalizadas quando as tarefas assíncronas são canceladas. Em quase todas as situações, a exceção deve ser levantada novamente.

Importante: This exception is a subclass of *Exception*, so it can be accidentally suppressed by an overly broad `try..except` block:

```
try:
    await operation
except Exception:
    # The cancellation is broken because the *except* block
    # suppresses the CancelledError exception.
    log.log('an error has occurred')
```

Instead, the following pattern should be used:

```
try:
    await operation
except asyncio.CancelledError:
```

(continua na próxima página)

(continuação da página anterior)

```
    raise
except Exception:
    log.log('an error has occurred')
```

exception asyncio.**InvalidStateError**
Estado interno inválido de *Task* ou *Future*.

Pode ser levantada em situações como definir um valor de resultado para um objeto *Future* que já tem um valor de resultado definido.

exception asyncio.**SendfileNotAvailableError**
A *syscall* "sendfile" não está disponível para o soquete ou tipo de arquivo fornecido.

Uma subclasse de *RuntimeError*.

exception asyncio.**IncompleteReadError**
A operação de leitura solicitada não foi totalmente concluída.

Levantada pelas *APIs de fluxo de asyncio*.

Esta exceção é uma subclasse de *EOFError*.

expected
O número total (*int*) de bytes esperados.

partial
Uma string de *bytes* lida antes do final do fluxo ser alcançado.

exception asyncio.**LimitOverrunError**
Atingiu o limite de tamanho do buffer ao procurar um separador.

Levantada pelas *APIs de fluxo de asyncio*.

consumed
O número total de bytes a serem consumidos.

19.1.7 Loop de Eventos

Prefácio

O loop de eventos é o núcleo de toda aplicação asyncio. Loops de evento executam tarefas e callbacks assíncronos, realizam operações de entrada e saída e executam subprocessos.

Os desenvolvedores de aplicação normalmente devem usar as funções asyncio de alto nível, como *asyncio.run()*, e devem raramente precisar fazer referência ao objeto de loop ou chamar seus métodos. Esta seção destina-se principalmente a autores de código de baixo nível, bibliotecas e frameworks, que precisam de um controle mais preciso sobre o comportamento do laço de evento.

Obtendo o Loop de Eventos

As seguintes funções baixo nível podem ser usadas para obter, definir, ou criar um laço de eventos:

asyncio.**get_running_loop**()
Retorna o laço de eventos em execução na thread atual do sistema operacional.

Se não existir nenhum laço de eventos em execução, um *RuntimeError* é levantado. Esta função somente pode ser chamada a partir de uma corrotina ou uma função de retorno.

Novo na versão 3.7.

asyncio.**get_event_loop**()
Obtém o laço de eventos atual.

Se não existe nenhum laço de eventos definido na thread atual do sistema operacional, é a thread principal do sistema operacional, e *set_event_loop()* ainda não foi chamada, asyncio irá criar um novo laço de eventos e defini-lo como o atual.

Devido ao fato desta função ter um comportamento particularmente complexo (especialmente quando políticas de laço de eventos customizadas estão sendo usadas), usar a função *get_running_loop()* é preferido ao invés de *get_event_loop()* em corrotinas e funções de retorno.

Considere também usar a função *asyncio.run()* ao invés de usar funções de baixo nível para manualmente criar e fechar um laço de eventos.

asyncio.**set_event_loop**(*loop*)
Define *loop* como o laço de eventos atual para a thread atual do sistema operacional.

asyncio.**new_event_loop**()
Cria um novo objeto de laço de eventos.

Perceba que o comportamento das funções *get_event_loop()*, *set_event_loop()*, e *new_event_loop()* podem ser alteradas *definindo uma política de laço de eventos customizada*.

Conteúdo

Esta página de documentação contém as seguintes seções:

- A seção *Métodos do laço de eventos* é a documentação de referência das APIs de laço de eventos;
- A seção *Tratadores de função de retorno* documenta as instâncias das classes *Handle* e *TimerHandle*, que são retornadas por métodos de agendamento tais como *loop.call_soon()* e *loop.call_later()*;
- A seção *Objetos Server* documenta tipos retornados a partir de métodos de laço de eventos, como *loop.create_server()*;
- A seção *Implementações do Laço de Eventos* documenta as classes *SelectorEventLoop* e *ProactorEventLoop*;
- A seção *Exemplos* demonstra como trabalhar com algumas APIs do laço de eventos APIs.

Métodos do laço de eventos

Laços de eventos possuem APIs de **baixo nível** para as seguintes situações:

Executar e interromper o laço

`loop.`**`run_until_complete`**`(`*future*`)`
Executar até que o *future* (uma instância da classe `Future`) seja completada.

Se o argumento é um *objeto corrotina*, ele é implicitamente agendado para executar como uma `asyncio.Task`.

Retorna o resultado do Future ou levanta sua exceção.

`loop.`**`run_forever`**`()`
Executa o laço de eventos até que `stop()` seja chamado.

Se `stop()` for chamado antes que `run_forever()` seja chamado, o laço irá pesquisar o seletor de E/S uma vez com um tempo limite de zero, executar todas as funções de retorno agendadas na resposta de eventos de E/S (e aqueles que já estavam agendados), e então sair.

Se `stop()` for chamado enquanto `run_forever()` estiver executando, o laço irá executar o lote atual de funções de retorno e então sair. Perceba que novas funções de retorno agendadas por funções de retorno não serão executadas neste caso; ao invés disso, elas serão executadas na próxima vez que `run_forever()` ou `run_until_complete()` forem chamadas.

`loop.`**`stop`**`()`
Para o laço de eventos.

loop.**is_running**()
Retorna `True` se o laço de eventos estiver em execução.

loop.**is_closed**()
Retorna `True` se o laço de eventos foi fechado.

loop.**close**()
Fecha o loop de eventos.

O laço não deve estar em execução quando esta função for chamada. Qualquer função de retorno pendente será descartada.

Este método limpa todas as filas e desliga o executor, mas não aguarda pelo encerramento do executor.

Este método é idempotente e irreversível. Nenhum outro método deve ser chamado depois que o laço de eventos esteja fechado.

coroutine loop.**shutdown_asyncgens**()
Agenda todos os objetos *geradores assíncronos* atualmente abertos para serem fechados com uma chamada `aclose()`. Após chamar este método, o laço de eventos emitirá um aviso se um novo gerador assíncrono for iterado. Isso deve ser utilizado para finalizar de forma confiável todos os geradores assíncronos agendados.

Perceba que não é necessário chamar esta função quando *`asyncio.run()`* for usado.

Exemplo:

```
try:
    loop.run_forever()
finally:
    loop.run_until_complete(loop.shutdown_asyncgens())
    loop.close()
```

Novo na versão 3.6.

Agendando funções de retorno

loop.**call_soon**(*callback*, **args*, *context=None*)
Schedule a *callback* to be called with *args* arguments at the next iteration of the event loop.

Funções de retorno são chamadas na ordem em que elas foram registradas. Cada função de retorno será chamada exatamente uma vez.

Um argumento opcional somente-nomeado *context* permite especificar um *`contextvars.Context`* customizado para executar na *função de retorno*. O contexto atual é usado quando nenhum *context* é fornecido.

Uma instância de *`asyncio.Handle`* é retornado, o qual pode ser usado posteriormente para cancelar a função de retorno.

Este método não é seguro para thread.

loop.**call_soon_threadsafe**(*callback*, **args*, *context=None*)
Uma variante segura para thread do *`call_soon()`*. Deve ser usada para agendar funções de retorno *a partir de outra thread*.

Veja a seção *concorrência e multithreading* da documentação.

Alterado na versão 3.7: O parâmetro somente-nomeado *context* foi adicionado. Veja **PEP 567** para mais detalhes.

Nota: Maior parte das funções de agendamento *`asyncio`* não permite passar argumentos nomeados. Para fazer isso, use *`functools.partial()`*:

```
# will schedule "print("Hello", flush=True)"
loop.call_soon(
    functools.partial(print, "Hello", flush=True))
```

Usar objetos parciais é usualmente mais conveniente que usar lambdas, pois asyncio pode renderizar objetos parciais melhor durante debug e mensagens de erro.

Agendando funções de retorno atrasadas

Laço de eventos fornece mecanismos para agendar funções de retorno para serem chamadas em algum ponto no futuro. Laço de eventos usa relógios monotônico para acompanhar o tempo.

`loop.`**`call_later`**(*delay*, *callback*, **args*, *context=None*)
Agenda *callback* para ser chamada após o *delay* número de segundos fornecido (pode ser um inteiro ou um ponto flutuante).

Uma instância de *asyncio.TimerHandle* é retornada, a qual pode ser usada para cancelar a função de retorno.

callback será chamada exatamente uma vez. Se duas funções de retorno são agendadas para exatamente o mesmo tempo, a ordem na qual elas são chamadas é indefinida.

O *args* posicional opcional será passado para a função de retorno quando ela for chamada. Se você quiser que a função de retorno seja chamada com argumentos nomeados, use *functools.partial()*.

Um argumento opcional somente-nomeado *context* permite especificar um *contextvars.Context* customizado para executar na *função de retorno*. O contexto atual é usado quando nenhum *context* é fornecido.

Alterado na versão 3.7: O parâmetro somente-nomeado *context* foi adicionado. Veja **PEP 567** para mais detalhes.

Alterado na versão 3.7.1: In Python 3.7.0 and earlier with the default event loop implementation, the *delay* could not exceed one day. This has been fixed in Python 3.7.1.

`loop.`**`call_at`**(*when*, *callback*, **args*, *context=None*)
Agenda *callback* para ser chamada no timestamp absoluto fornecido *when* (um inteiro ou um ponto flutuante), usando o mesmo horário de referência que *loop.time()*.

O comportamento deste método é o mesmo que *call_later()*.

Uma instância de *asyncio.TimerHandle* é retornada, a qual pode ser usada para cancelar a função de retorno.

Alterado na versão 3.7: O parâmetro somente-nomeado *context* foi adicionado. Veja **PEP 567** para mais detalhes.

Alterado na versão 3.7.1: In Python 3.7.0 and earlier with the default event loop implementation, the difference between *when* and the current time could not exceed one day. This has been fixed in Python 3.7.1.

`loop.`**`time`**()
Retorna o horário atual, como um valor *float*, de acordo com o relógio monotônico interno do laço de eventos.

Nota: Alterado na versão 3.8: No Python 3.7 e anterior, tempos limites (*delay* relativo ou *when* absoluto) não poderiam exceder um dia. Isto foi ajustado no Python 3.8.

Ver também:

A função *asyncio.sleep()*.

Criando Futures e Tasks

`loop.`**`create_future`**`()`
Cria um objeto *asyncio.Future* atachado ao laço de eventos.

Este é o modo preferido para criar Futures em asyncio. Isto permite que laços de eventos de terceiros forneçam implementações alternativas do objeto Future (com melhor desempenho ou instrumentação).

Novo na versão 3.5.2.

`loop.`**`create_task`**`(`*coro*`)`
Agenda a execução de uma *Coroutines*. Retorna um objeto *Task*.

Laços de eventos de terceiros podem usar suas próprias subclasses de *Task* para interoperabilidade. Neste caso, o tipo do resultado é uma subclasse de *Task*.

`loop.`**`set_task_factory`**`(`*factory*`)`
Define a factory da tarefa que será usada por *loop.create_task()*.

Se *factory* for `None`, a factory da task padrão será definida. Caso contrário, *factory* deve ser algo *chamável* com a assinatura coincidindo com `(loop, coro)`, onde *loop* é uma referência para o laço de eventos ativo, e *coro* é um objeto corrotina. O objeto chamável deve retornar um objeto compatível com *asyncio.Future*.

`loop.`**`get_task_factory`**`()`
Retorna uma factory de tarefa ou `None` se a factory padrão estiver em uso.

Abrindo conexões de rede

`coroutine` `loop.`**`create_connection`**`(`*protocol_factory*, *host=None*, *port=None*, ***, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*`)`
Abre uma conexão de transporte para streaming, para um endereço fornecido, especificado por *host* e *port*.

A família de soquetes pode ser *AF_INET* ou *AF_INET6* dependendo do *host* (ou do argumento *family*, se fornecido).

O tipo de soquete será *SOCK_STREAM*.

protocol_factory deve ser um chamável que retorne uma implementação do *protocolo asyncio*.

Este método tentará estabelecer a conexão em segundo plano. Quando tiver sucesso, ele retorna um par `(transport, protocol)`.

A sinopse cronológica da operação subjacente é conforme abaixo:

1. A conexão é estabelecida e um *transporte* é criado para ela.
2. *protocol_factory* é chamada sem argumentos e é esperada que retorne uma instância de *protocolo*.
3. A instância de protocolo é acoplada com o transporte, através da chamada do seu método *connection_made()*.
4. Uma tupla `(transport, protocol)` é retornada ao ter sucesso.

O transporte criado é um stream bi-direcional dependente de implementação.

Outros argumentos:

- *ssl*: se fornecido e não for falso, um transporte SSL/TLS é criado (por padrão um transporte TCP simples é criado). Se *ssl* for um objeto *ssl.SSLContext*, este contexto é usado para criar o transporte; se *ssl* for *True*, um contexto padrão retornado de *ssl.create_default_context()* é usado.

  Ver também:

SSL/TLS security considerations

- *server_hostname* define ou substitui o hostname que o certificado do servidor de destino será pareado contra. Deve ser passado apenas se *ssl* não for `None`. Por padrão o valor do argumento *host* é usado. Se *host* for vazio, não existe valor padrão e você deve passar um valor para *server_hostname*. Se *server_hostname* for uma string vazia, o pareamento de hostname é desabilitado (o que é um risco de segurança sério, permitindo ataques potenciais man-in-the-middle).
- *family*, *proto*, *flags* são os endereços familiares, protocolos e sinalizadores opcionais a serem passados por getaddrinfo() para resolução do *host*. Se fornecidos, eles devem ser todos inteiros e constantes correspondentes do módulo `socket`.
- *sock*, if given, should be an existing, already connected `socket.socket` object to be used by the transport. If *sock* is given, none of *host*, *port*, *family*, *proto*, *flags* and *local_addr* should be specified.
- *local_addr*, if given, is a `(local_host, local_port)` tuple used to bind the socket to locally. The *local_host* and *local_port* are looked up using `getaddrinfo()`, similarly to *host* and *port*.
- *ssl_handshake_timeout* é (para uma conexão TLS) o tempo em segundos para aguardar pelo encerramento do aperto de mão TLS, antes de abortar a conexão. `60.0` segundos se for"None" (valor padrão).

Novo na versão 3.7: O parâmetro *ssl_handshake_timeout*.

Alterado na versão 3.6: A opção de soquete `TCP_NODELAY` é definida por padrão para todas as conexões TCP.

Alterado na versão 3.5: Adicionado suporte para SSL/TLS na `ProactorEventLoop`.

Ver também:

A função `open_connection()` é uma API alternativa de alto nível. Ela retorna um par de (`StreamReader`, `StreamWriter`) que pode ser usado diretamente em código async/await.

coroutine loop.**create_datagram_endpoint**(*protocol_factory*, *local_addr=None*, *remote_addr=None*, ***, *family=0*, *proto=0*, *flags=0*, *reuse_address=None*, *reuse_port=None*, *allow_broadcast=None*, *sock=None*)

Nota: O parâmetro *reuse_address* não é mais suportado, assim como usar `SO_REUSEADDR` representa uma preocupação de segurança significativa para UDP. Passar `reuse_address=True` explicitamente irá levantar uma exceção.

When multiple processes with differing UIDs assign sockets to an indentical UDP socket address with `SO_REUSEADDR`, incoming packets can become randomly distributed among the sockets.

Para plataformas suportadas, *reuse_port* pode ser usado como um substituto para funcionalidades similares. Com *reuse_port*, `SO_REUSEPORT` é usado ao invés, o qual especificamente previne processos com diferentes UIDs de atribuir soquetes para o mesmo endereço do soquete.

Cria uma conexão de datagrama.

A família de soquetes pode ser `AF_INET`, `AF_INET6`, ou `AF_UNIX`, dependendo do *host* (ou do argumento *family*, se fornecido).

O tipo de soquete será `SOCK_DGRAM`.

protocol_factory deve ser algo chamável, retornando uma implementação de *protocolo*.

Uma tupla de `(transport, protocol)` é retornada em caso de sucesso.

Outros argumentos:

- *local_addr*, if given, is a `(local_host, local_port)` tuple used to bind the socket to locally. The *local_host* and *local_port* are looked up using `getaddrinfo()`.
- *remote_addr*, se fornecido, é uma tupla de `(remote_host, remote_port)` usada para conectar o soquete a um endereço remoto. O *remote_host* e a *remote_port* são procurados usando `getaddrinfo()`.
- *family*, *proto*, *flags* são os endereços familiares, protocolo e flags opcionais a serem passados para `getaddrinfo()` para resolução do *host*. Se fornecido, esses devem ser todos inteiros do módulo de constantes `socket` correspondente.
- *reuse_port* avisa o kernel para permitir este endpoint para ser ligado a mesma porta da mesma forma que outros endpoints existentes estão ligados a, contanto que todos eles definam este flag quando forem criados. Esta opção não é suportada no Windows e em alguns sistemas Unix. Se a constante `SO_REUSEPORT` não estiver definida, então esta capacidade não é suportada.
- *allow_broadcast* avisa o kernel para permitir que este endpoint envie mensagens para o endereço de broadcast.
- *sock* pode opcionalmente ser especificado em ordem para usar um objeto `socket.socket` pre-existente, já conectado, para ser usado pelo transporte. Se especificado, *local_addr* e *remote_addr* devem ser omitidos (devem ser `None`).

On Windows, with `ProactorEventLoop`, this method is not supported.

Veja *protocolo UDP eco cliente* e *protocolo UDP eco servidor* para exemplos.

Alterado na versão 3.4.4: Os parâmetros *family*, *proto*, *flags*, *reuse_address*, *reuse_port, *allow_broadcast*, e *sock* foram adicionados.

Alterado na versão 3.7.6: O parâmetro *reuse_address* não é mais suportado devido a preocupações de segurança.

coroutine loop.**create_unix_connection**(*protocol_factory*, *path=None*, *, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*)

Cria uma conexão Unix.

A família de soquete será `AF_UNIX`; o tipo de soquete será `SOCK_STREAM`.

Uma tupla de `(transport, protocol)` é retornada em caso de sucesso.

path é o nome de um soquete de domínio Unix e é obrigatório, a não ser que um parâmetro *sock* seja esecificado. Soquetes Unix abstratos, `str`, `bytes`, e caminhos `Path` são suportados.

Veja a documentação do método `loop.create_connection()` para informações a respeito de argumentos para este método.

Disponibilidade: Unix.

Novo na versão 3.7: O parâmetro *ssl_handshake_timeout*.

Alterado na versão 3.7: O parâmetro *path* agora pode ser um *objeto caminho ou similar*.

Criando servidores de rede

coroutine loop.**create_server**(*protocol_factory*, *host=None*, *port=None*, *, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *ssl_handshake_timeout=None*, *start_serving=True*)

Cria um servidor TCP (tipo de soquete `SOCK_STREAM`) escutando em *port* do entedeço *host*.

Retorna um objeto `Server`.

Argumentos:

- *protocol_factory* deve ser algo chamável, retornando uma implementação de *protocolo*.
- O parâmetro *host* pode ser definido para diversos tipos, o qual determina onde o servidor deve escutar:
  - Se *host* for uma string, o servidor TCP está vinculado a apenas uma interface de rede, especificada por *host*.
  - Se *host* é uma sequência de strings, o servidor TCP está vinculado a todas as interfaces de rede especificadas pela sequência.
  - Se *host* é uma string vazia ou `None`, todas as interfaces são assumidas e uma lista de múltiplos soquetes será retornada (muito provavelmente um para IPv4 e outro para IPv6).
- *family* pode ser definido para `socket.AF_INET` ou `AF_INET6` para forçar o soquete a usar IPv4 ou IPv6. Se não for definido, *family* será determinado a partir do nome do servidor (por padrão será `AF_UNSPEC`).
- *flags* é uma máscara de bits para `getaddrinfo()`.
- *sock* pode opcionalmente ser especificado para usar um objeto soquete pré-existente. Se especificado, *host* e *port* não devem ser especificados.
- *backlog* é o número máximo de conexões enfileiradas pasadas para `listen()` (padrão é 100).
- *ssl* pode ser definido para uma instância de `SSLContext` para habilitar TLS sobre as conexões aceitas.
- *reuse_address* diz ao kernel para reusar um soquete local em estado `TIME_WAIT`, serm aguardar pela expiração natural do seu tempo limite. Se não especificado, será automaticamente definida como `True` no Unix.
- *reuse_port* diz ao kernel para permitir que este endpoint seja vinculado a mesma porta que outros endpoints existentes estão vinculados, contanto que todos eles definam este sinalizador quando forem criados. Esta opção não é suportada no Windows.
- *ssl_handshake_timeout* é (para um servidor TLS) o tempo em segundos para aguardar pelo aperto de mão TLS ser concluído, antes de abortar a conexão. `60.0` segundos se `None` (valor padrão).
- Definir *start_serving* para `True` (o valor padrão) faz o servidor criado começar a aceitar conexões imediatamente. Quando definido para `False`, o usuário deve aguardar com `Server.start_serving()` ou `Server.serve_forever()` para fazer o servidor começar a aceitar conexões.

Novo na versão 3.7: Adicionado os parâmetros *ssl_handshake_timeout* e *start_serving*.

Alterado na versão 3.6: A opção de soquete `TCP_NODELAY` é definida por padrão para todas as conexões TCP.

Alterado na versão 3.5: Adicionado suporte para SSL/TLS na `ProactorEventLoop`.

Alterado na versão 3.5.1: O parâmetro *host* pode ser uma sequência de strings.

Ver também:

A função `start_server()` é uma API alternativa de alto nível que retorna um par de `StreamReader` e `StreamWriter` que pode ser usado em um código async/await.

coroutine loop.**create_unix_server**(*protocol_factory*, *path=None*, ***, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *start_serving=True*)

Similar a `loop.create_server()`, mas trabalha com a familia de soquete `AF_UNIX`.

path é o nome de um soquete de domínio Unix, e é obrigatório, a não ser que um argumento *sock* seja fornecido. Soquetes Unix abstratos, `str`, `bytes`, e caminhos `Path` são suportados.

Veja a documentação do método `loop.create_server()` para informações sobre argumentos para este método.

Disponibilidade: Unix.

Novo na versão 3.7: Os parâmetros *ssl_handshake_timeout* e *start_serving*.

Alterado na versão 3.7: O parâmetro *path* agora pode ser um objeto `Path`.

coroutine loop.**connect_accepted_socket**(*protocol_factory*, *sock*, *, *ssl=None*, *ssl_handshake_timeout=None*)

Envolve uma conexão já aceita em um par transporte/protocolo.

Este método pode ser usado por servidores que aceitam conexões fora do asyncio, mas que usam asyncio para manipulá-las.

Parâmetros:

- *protocol_factory* deve ser algo chamável, retornando uma implementação de *protocolo*.
- *sock* é um objeto soquete pré-existente retornado a partir de `socket.accept`.
- *ssl* pode ser definido para um `SSLContext` para habilitar SSL sobre as conexões aceitas.
- *ssl_handshake_timeout* é (para uma conexão SSL) o tempo em segundos para aguardar pelo aperto de mão SSL ser concluído, antes de abortar a conexão. `60.0` segundos se `None` (valor padrão).

Retorna um par `(transport, protocol)`.

Novo na versão 3.7: O parâmetro *ssl_handshake_timeout*.

Novo na versão 3.5.3.

Transferindo arquivos

coroutine loop.**sendfile**(*transport*, *file*, *offset=0*, *count=None*, *, *fallback=True*)

Envia um *file* sobre um *transport*. Retorna o número total de bytes enviados.

O método usa `os.sendfile()` de alto desempenho, se disponível.

file deve ser um objeto arquivo regular aberto em modo binário.

offset indica a partir de onde deve iniciar a leitura do arquivo. Se especificado, *count* é o número total de bytes para transmitir, ao contrário de transmitir o arquivo até que EOF seja atingido. A posição do arquivo é sempre atualizada, mesmo quando este método levanta um erro, e `file.tell()` pode ser usado para obter o número atual de bytes enviados.

fallback definido para `True` faz o asyncio manualmente ler e enviar o arquivo quando a plataforma não suporta a chamada de sistema sendfile (por exemplo Windows ou soquete SSL no Unix).

Levanta `SendfileNotAvailableError` se o sistema não suporta a chamada de sistema *sendfile* e *fallback* é `False`.

Novo na versão 3.7.

Atualizando TLS

coroutine loop.**start_tls**(*transport*, *protocol*, *sslcontext*, *, *server_side=False*, *server_hostname=None*, *ssl_handshake_timeout=None*)

Atualiza um conexão baseada em transporte existente para TLS.

Retorna uma nova instância de transporte, que o *protocol* deve começar a usar imediatamente após o *await*. A instância de *transport* passada para o método *start_tls* nunca deve ser usada novamente.

Parâmetros:

- instâncias de *transport* e *protocol*, que métodos como `create_server()` e `create_connection()` retornam.
- *sslcontext*: uma instância configurada de `SSLContext`.
- *server_side* informe `True` quando uma conexão no lado do servidor estiver sendo atualizada (como a que é criada por `create_server()`).
- *server_hostname*: define ou substitui o nome do host no qual o servidor alvo do certificado será comparado.
- *ssl_handshake_timeout* é (para uma conexão TLS) o tempo em segundos para aguardar pelo encerramento do aperto de mão TLS, antes de abortar a conexão. `60.0` segundos se for"None" (valor padrão).

Novo na versão 3.7.

Observando descritores de arquivo

`loop.`**`add_reader`**(*fd*, *callback*, **args*)
Começa a monitorar o descritor de arquivo *fd* para disponibilidade de leitura e invoca a *callback* com os argumentos especificados assim que *fd* esteja disponível para leitura.

`loop.`**`remove_reader`**(*fd*)
Para de monitorar o descritor de arquivo *fd* para disponibilidade de leitura.

`loop.`**`add_writer`**(*fd*, *callback*, **args*)
Começa a monitorar o descritor de arquivo *fd* para disponibilidade de escrita e invoca a *callback* com os argumentos especificados assim que *fd* esteja disponível para escrita.

Use *`functools.partial()` para passar argumentos nomeados* para a *callback*.

`loop.`**`remove_writer`**(*fd*)
Para de monitorar o descritor de arquivo *fd* para disponibilidade de escrita.

Veja também a seção de *Suporte a Plataformas* para algumas limitações desses métodos.

Trabalhando com objetos soquete diretamente

Em geral, implementações de protocolo que usam APIs baseadas em transporte, tais como `loop.create_connection()` e `loop.create_server()` são mais rápidas que implementações que trabalham com soquetes diretamente. Entretanto, existem alguns casos de uso quando o desempenho não é crítica, e trabalhar com objetos `socket` diretamente é mais conveniente.

`coroutine` `loop.`**`sock_recv`**(*sock*, *nbytes*)
Recebe até *nbytes* do *sock*. Versão assíncrona de `socket.recv()`.

Retorna os dados recebidos como um objeto de bytes.

sock deve ser um soquete não bloqueante.

Alterado na versão 3.7: Apesar deste método sempre ter sido documentado como um método de corrotina, versões anteriores ao Python 3.7 retornavam um `Future`. Desde o Python 3.7 este é um método `async def`.

`coroutine` `loop.`**`sock_recv_into`**(*sock*, *buf*)
Dados recebidos do *sock* no buffer *buf*. Modelado baseado no método bloqueante `socket.recv_into()`.

Retorna o número de bytes escritos no buffer.

sock deve ser um soquete não bloqueante.

Novo na versão 3.7.

coroutine loop.**sock_sendall**(*sock*, *data*)
Envia *data* para o soquete *sock*. Versão assíncrona de *socket.sendall()*.

Este método continua a enviar para o soquete até que todos os dados em *data* tenham sido enviados ou um erro ocorra. None é retornado em caso de sucesso. Ao ocorrer um erro, uma exceção é levantada. Adicionalmente, não existe nenhuma forma de determinar quantos dados, se algum, foram processados com sucesso pelo destinatário na conexão.

sock deve ser um soquete não bloqueante.

Alterado na versão 3.7: Apesar deste método sempre ter sido documentado como um método de corrotina, antes do Python 3.7 ele retornava um *Future*. A partir do Python 3.7, este é um método async def.

coroutine loop.**sock_connect**(*sock*, *address*)
Conecta o *sock* em um endereço *address* remoto.

Versão assíncrona de *socket.connect()*.

sock deve ser um soquete não bloqueante.

Alterado na versão 3.5.2: address não precisa mais ser resolvido. sock_connect irá tentar verificar se *address* já está resolvido chamando *socket.inet_pton()*. Se não estiver, *loop.getaddrinfo()* será usado para resolver *address*.

Ver também:

loop.create_connection() e *asyncio.open_connection()*.

coroutine loop.**sock_accept**(*sock*)
Aceita uma conexão. Modelado baseado no método bloqueante *socket.accept()*.

O soquete deve estar vinculado a um endereço e escutando por conexões. O valor de retorno é um par (conn, address) onde *conn* é um *novo* objeto de soquete usável para enviar e receber dados na conexão, e *address* é o endereço vinculado ao soquete no outro extremo da conexão.

sock deve ser um soquete não bloqueante.

Alterado na versão 3.7: Apesar deste método sempre ter sido documentado como um método de corrotina, antes do Python 3.7 ele retornava um *Future*. Desde o Python 3.7, este é um método async def.

Ver também:

loop.create_server() e *start_server()*.

coroutine loop.**sock_sendfile**(*sock*, *file*, *offset=0*, *count=None*, ***, *fallback=True*)
Envia um arquivo usando *os.sendfile* de alto desempenho se possível. Retorna o número total de bytes enviados.

Versão assíncrona de *socket.sendfile()*.

sock deve ser um *socket* *socket.SOCK_STREAM* não bloqueante.

file deve ser um objeto arquivo regular aberto em modo binário.

offset indica a partir de onde deve iniciar a leitura do arquivo. Se especificado, *count* é o número total de bytes para transmitir, ao contrário de transmitir o arquivo até que EOF seja atingido. A posição do arquivo é sempre atualizada, mesmo quando este método levanta um erro, e *file.tell()* pode ser usado para obter o número atual de bytes enviados.

fallback, quando definido para True, faz asyncio ler e enviar manualmente o arquivo, quando a plataforma não suporta a chamada de sistema sendfile (por exemplo Windows ou soquete SSL no Unix).

Levanta *SendfileNotAvailableError* se o sistema não suporta chamadas de sistema *sendfile* e *fallback* é False.

sock deve ser um soquete não bloqueante.

Novo na versão 3.7.

DNS

coroutine loop.**getaddrinfo**(*host*, *port*, *, *family=0*, *type=0*, *proto=0*, *flags=0*)
Versão assíncrona de `socket.getaddrinfo()`.

coroutine loop.**getnameinfo**(*sockaddr*, *flags=0*)
Versão assíncrona de `socket.getnameinfo()`.

Alterado na versão 3.7: Ambos os métodos *getaddrinfo* e *getnameinfo* sempre foram documentados para retornar uma corrotina, mas antes do Python 3.7 eles estavam, na verdade, retornando objetos `asyncio.Future`. A partir do Python 3.7, ambos os métodos são corrotinas.

Trabalhando com encadeamentos

coroutine loop.**connect_read_pipe**(*protocol_factory*, *pipe*)
Registra o extremo da leitura de um *pipe* no laço de eventos.

protocol_factory deve ser um chamável que retorne uma implementação do *protocolo asyncio*.

pipe é um *objeto arquivo ou similar*.

Retorna um par `(transport, protocol)`, onde *transport* suporta a interface `ReadTransport` e *protocol* é um objeto instanciado pelo *protocol_factory*.

Com o `SelectorEventLoop` do laço de eventos, o *pipe* é definido para modo não bloqueante.

coroutine loop.**connect_write_pipe**(*protocol_factory*, *pipe*)
Registra o extremo de escrita do *pipe* no laço de eventos.

protocol_factory deve ser um chamável que retorne uma implementação do *protocolo asyncio*.

pipe é um *objeto arquivo ou similar*.

Retorna um part `(transport, protocol)`, onde *transport* suporta a interface `WriteTransport` e *protocol* é um objeto instanciado pelo *protocol_factory*.

Com o `SelectorEventLoop` do laço de eventos, o *pipe* é definido para modo não bloqueante.

Nota: `SelectorEventLoop` não suporta os métodos acima no Windows. Use `ProactorEventLoop` ao invés para Windows.

Ver também:

Os métodos `loop.subprocess_exec()` e `loop.subprocess_shell()`.

Sinais Unix

`loop.`**`add_signal_handler`**(*signum*, *callback*, **args*)
Define *callback* como o tratador para o sinal *signum*.

A função de retorno será invocada pelo *loop*, juntamente com outras funções de retorno enfileiradas e corrotinas executáveis daquele laço de eventos. Ao contrário de tratadores de sinal registrados usando `signal.signal()`, uma função de retorno registrada com esta função tem autorização para interagir com o laço de eventos.

Levanta `ValueError` se o número do sinal é inválido ou impossível de capturar. Levanta `RuntimeError` se existe um problema definindo o tratador.

Use *`functools.partial()` para passar argumentos nomeados* para a *callback*.

Assim como `signal.signal()`, esta função deve ser invocada na thread principal.

`loop.`**`remove_signal_handler`**(*sig*)
Remove o tratador para o sinal *sig*.

Retorna `True` se o tratador de sinal foi removido, ou `False` se nenhum tratador foi definido para o sinal fornecido.

Disponibilidade: Unix.

Ver também:

O módulo `signal`.

Executando código em conjuntos de threads ou processos

`awaitable` `loop.`**`run_in_executor`**(*executor*, *func*, **args*)
Providencia para a *func* ser chamada no executor especificado.

O argumento *executor* deve ser uma instância `concurrent.futures.Executor`. O executor padrão é usado se *executor* for `None`.

Exemplo:

```
import asyncio
import concurrent.futures

def blocking_io():
    # File operations (such as logging) can block the
    # event loop: run them in a thread pool.
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

def cpu_bound():
    # CPU-bound operations will block the event loop:
    # in general it is preferable to run them in a
    # process pool.
    return sum(i * i for i in range(10 ** 7))

async def main():
    loop = asyncio.get_running_loop()

    ## Options:

    # 1. Run in the default loop's executor:
    result = await loop.run_in_executor(
```

(continua na próxima página)

(continuação da página anterior)

```
        None, blocking_io)
    print('default thread pool', result)

    # 2. Run in a custom thread pool:
    with concurrent.futures.ThreadPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, blocking_io)
        print('custom thread pool', result)

    # 3. Run in a custom process pool:
    with concurrent.futures.ProcessPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom process pool', result)

asyncio.run(main())
```

Este método retorna um objeto *asyncio.Future*.

Use *functools.partial() para passar argumentos nomeados* para *func*.

Alterado na versão 3.5.3: *loop.run_in_executor()* não configura mais o atributo `max_workers` do executor do conjunto de thread que ele cria, ao invés disso ele deixa para o executor do conjunto de thread (*ThreadPoolExecutor*) para setar o valor padrão.

`loop.`**`set_default_executor`**(*executor*)

Define *executor* como o executor padrão usado por *run_in_executor()*. *executor* deve ser uma instância de *ThreadPoolExecutor*.

Obsoleto desde a versão 3.7: Usar um executor que não é uma instância de *ThreadPoolExecutor* foi descontinuado, e irá disparar um erro no Python 3.9.

executor deve ser uma instância de *concurrent.futures.ThreadPoolExecutor*.

Tratando erros da API

Permite customizar como exceções são tratadas no laço de eventos.

`loop.`**`set_exception_handler`**(*handler*)

Define *handler* como o novo tratador de exceções do laço de eventos.

Se *handler* for `None`, o tratador de exceções padrão será definido. Caso contrário, *handler* deve ser um chamável com a assinatura combinando `(loop, context)`, onde `loop` é a referência para o laço de eventos ativo, e `context` é um objeto `dict` contendo os detalhes da exceção (veja a documentação *call_exception_handler()* para detalhes a respeito do contexto).

`loop.`**`get_exception_handler`**()

Retorna o tratador de exceção atual, ou `None` se nenhum tratador de exceção customizado foi definido.

Novo na versão 3.5.2.

`loop.`**`default_exception_handler`**(*context*)

Tratador de exceção padrão.

Isso é chamado quando uma exceção ocorre e nenhum tratador de exceção foi definido. Isso pode ser chamado por um tratador de exceção customizado que quer passar adiante para o comportamento do tratador padrão.

parâmetro *context* tem o mesmo significado que em *call_exception_handler()*.

loop.**call_exception_handler**(*context*)
Chama o tratador de exceção do laço de eventos atual.

context é um objeto `dict` contendo as seguintes chaves (novas chaves podem ser introduzidas em versões futuras do Python):

- 'message': Mensagem de erro;
- 'exception' (opcional): Objeto Exception;
- 'future' (opcional): instância de *asyncio.Future*;
- 'handle' (opcional): instância de *asyncio.Handle*;
- 'protocol' (opcional): instância de *Protocol*;
- 'transport' (opcional): instância de *Transport*;
- 'socket' (optional): *socket.socket* instance.

Nota: Este método não deve ser substituído em subclasses de laços de evento. Para tratamento de exceções customizadas, use o método *set_exception_handler()*.

Habilitando o modo de debug

loop.**get_debug**()
Obtém o modo de debug (*bool*) do laço de eventos.

O valor padrão é `True` se a variável de ambiente `PYTHONASYNCIODEBUG` estiver definida para uma string não vazia, `False` caso contrário.

loop.**set_debug**(*enabled: bool*)
Define o modo de debug do laço de eventos.

Alterado na versão 3.7: The new `-X dev` command line option can now also be used to enable the debug mode.

Ver também:

O *modo de debug de asyncio*.

Executando Subprocessos

Métodos descritos nestas sub-seções são de baixo nível. Em código async/await regular, considere usar as funções convenientes de alto nível *asyncio.create_subprocess_shell()* e *asyncio.create_subprocess_exec()* ao invés.

Nota: O laço de eventos asyncio padrão no **Windows** não suporta subprocessos. Veja *Suporte a Subprocesso no Windows* para detalhes.

coroutine loop.**subprocess_exec**(*protocol_factory*, **args*, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)
Cria um subprocesso a partir de um ou mais argumentos de string especificados por *args*.

args deve ser uma lista de strings representada por:

- *str*;

- ou *bytes*, encodados na *codificação do sistema de arquivos*.

A primeira string especifica o programa executável, e as strings remanescentes especificam os argumentos. Juntas, argumentos em string formam o `argv` do programa.

Isto é similar a classe *subprocess.Popen* da biblioteca padrão ser chamada com `shell=False` e a lista de strings ser passada como o primeiro argumento; entretanto, onde *Popen* recebe apenas um argumento no qual é uma lista de strings, *subprocess_exec* recebe múltiplos argumentos string.

O *protocol_factory* deve ser um chamável que retorne uma subclasse da classe *asyncio.SubprocessProtocol*.

Outros parâmetros:

- *stdin*: either a file-like object representing a pipe to be connected to the subprocess's standard input stream using *connect_write_pipe()*, or the *subprocess.PIPE* constant (default). By default a new pipe will be created and connected.
- *stdout*: either a file-like object representing the pipe to be connected to the subprocess's standard output stream using *connect_read_pipe()*, or the *subprocess.PIPE* constant (default). By default a new pipe will be created and connected.
- *stderr*: either a file-like object representing the pipe to be connected to the subprocess's standard error stream using *connect_read_pipe()*, or one of *subprocess.PIPE* (default) or *subprocess.STDOUT* constants.

  By default a new pipe will be created and connected. When *subprocess.STDOUT* is specified, the subprocess' standard error stream will be connected to the same pipe as the standard output stream.
- All other keyword arguments are passed to *subprocess.Popen* without interpretation, except for *bufsize*, *universal_newlines* and *shell*, which should not be specified at all.

Veja o construtor da classe *subprocess.Popen* para documentação sobre outros argumentos.

Retorna um par `(transport, protocol)`, onde *transport* conforma com a classe base *asyncio.SubprocessTransport* e *protocol* é um objeto instanciado pelo *protocol_factory*.

coroutine loop.**subprocess_shell**(*protocol_factory*, *cmd*, ***, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)

Cria um subprocesso a partir do *cmd*, o qual pode ser um *str* ou uma string de *bytes* codificada na *codificação do sistema de arquivos*, usando a sintaxe "shell" da plataforma.

Isto é similar a classe *subprocess.Popen* da biblioteca padrão sendo chanada com `shell=True`.

O argumento *protocol_factory* deve ser um chamável que retorna uma subclasse da classe *SubprocessProtocol*.

Veja *subprocess_exec()* para mais detalhes sobre os argumentos remanescentes.

Retorna um par `(transport, protocol)`, onde *transport* conforma com a classe base *SubprocessTransport* e *protocol* é um objeto instanciado pelo *protocol_factory*.

Nota: É responsabilidade da aplicação garantir que todos os espaços em branco e caracteres especiais sejam tratados apropriadamente para evitar vulnerabilidades de injeção shell . A função *shlex.quote()* pode ser usada para escapar espaços em branco e caracteres especiais apropriadamente em strings que serão usadas para construir comandos shell.

Tratadores de função de retorno

class asyncio.**Handle**
Um objeto empacotador de função de retorno retornado por *loop.call_soon()*, *loop.call_soon_threadsafe()*.

cancel()
Cancela a função de retorno. Se a função de retorno já tiver sido cancelada ou executada, este método não tem efeito.

cancelled()
Retorna `True` se a função de retorno foi cancelada.

Novo na versão 3.7.

class asyncio.**TimerHandle**
Um objeto empacotador de função de retorno retornado por *loop.call_later()*, e *loop.call_at()*.

Esta classe é uma subclasse de *Handle*.

when()
Retorna o tempo de uma função de retorno agendada como *float* segundos.

O tempo é um timestamp absoluto, usando a mesma referência de tempo que *loop.time()*.

Novo na versão 3.7.

Objetos Server

Objetos Server são criados pelas funções *loop.create_server()*, *loop.create_unix_server()*, *start_server()*, e *start_unix_server()*.

Não instancie a classe diretamente

class asyncio.**Server**
Objetos *Server* são gerenciadores de contexto assíncronos. Quando usados em uma instrução `async with`, é garantido que o objeto Server está fechado e não está aceitando novas conexões quando a instrução `async with` estiver completa:

```
srv = await loop.create_server(...)

async with srv:
    # some code

# At this point, srv is closed and no longer accepts new connections.
```

Alterado na versão 3.7: Objeto Server é um gerenciador de contexto assíncrono desde o Python 3.7.

close()
Para de servir: fecha soquetes que estavam ouvindo e define o atributo *sockets* para `None`.

Os soquetes que representam conexões de clientes existentes que estão chegando são deixados em aberto.

O servidor é fechado de forma assíncrona, use a corrotina *wait_closed()* para aguartar até que o servidor esteja fechado.

get_loop()
Retorna o laço de eventos associado com o objeto server.

Novo na versão 3.7.

coroutine start_serving()
Começa a aceitar conexões.

Este método é method is idempotente, então ele pode ser cancelado quando o servidor já estiver servindo.

O parâmetro somente-nomeado *start_serving* para *loop.create_server()* e *asyncio.start_server()* permite criar um objeto Server que não está aceitando conexões inicialmente. Neste caso `Server.start_serving()`, ou *Server.serve_forever()* podem ser usados para fazer o Server começar a aceitar conexões.

Novo na versão 3.7.

coroutine serve_forever()
Começa a aceitar conexões até que a corrotina seja cancelada. Cancelamento da task "serve_forever"causa o fechamento do servidor.

Este método pode ser chamado se o servidor já estiver aceitando conexões. Apenas uma task"serve_forever" pode existir para cada objeto *Server*.

Exemplo:

```
async def client_connected(reader, writer):
    # Communicate with the client with
    # reader/writer streams.  For example:
    await reader.readline()

async def main(host, port):
    srv = await asyncio.start_server(
        client_connected, host, port)
    await srv.serve_forever()

asyncio.run(main('127.0.0.1', 0))
```

Novo na versão 3.7.

is_serving()
Retorna `True` se o servidor estiver aceitando novas conexões.

Novo na versão 3.7.

coroutine wait_closed()
Aguarda até o método *close()* completar.

sockets
List of *socket.socket* objects the server is listening on, or `None` if the server is closed.

Alterado na versão 3.7: Antes do Python 3.7 `Server.sockets` era usado para retornar uma lista interna de soquetes do server diretamente. No uma cópia dessa lista é retornada.

Implementações do Laço de Eventos

asyncio vem com duas implementações de laço de eventos diferente: *SelectorEventLoop* e *ProactorEventLoop*.

By default asyncio is configured to use *SelectorEventLoop* on all platforms.

class asyncio.**SelectorEventLoop**
Um laço de eventos baseado no módulo *selectors*.

Usa o *seletor* mais eficiente disponível para a plataforma fornecida. Também é possível configurar manualmente a implementação exata do seletor a ser utilizada:

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

Disponibilidade: Unix, Windows.

class asyncio.**ProactorEventLoop**

Um laço de eventos para Windows que usa "Conclusão de Portas I/O" (IOCP).

Disponibilidade: Windows.

An example how to use *ProactorEventLoop* on Windows:

```
import asyncio
import sys

if sys.platform == 'win32':
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
```

Ver também:

Documentação da MSDN sobre conclusão de portas I/O.

class asyncio.**AbstractEventLoop**

Classe base abstrata para laços de eventos compatíveis com asyncio.

A seção *Métodos de laço de eventos* lista todos os métodos que uma implementação alternativa de `AbstractEventLoop` deve definir.

Exemplos

Perceba que todos os exemplos nesta seção **propositalmente** mostram como usar as APIs de baixo nível do laço de eventos, tais como *loop.run_forever()* e *loop.call_soon()*. Aplicações asyncio modernas raramente precisam ser escritas desta forma; considere usar as funções de alto nível como *asyncio.run()*.

Hello World com call_soon()

Um exemplo usando o método *loop.call_soon()* para agendar uma função de retorno. A função de retorno exibe `"Hello World"` e então para o laço de eventos:

```
import asyncio

def hello_world(loop):
    """A callback to print 'Hello World' and stop the event loop"""
    print('Hello World')
    loop.stop()

loop = asyncio.get_event_loop()

# Schedule a call to hello_world()
loop.call_soon(hello_world, loop)
```

(continua na próxima página)

(continuação da página anterior)

```
# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

Ver também:

Um exemplo similar a *Hello World* criado com uma corrotina e a função `run()`.

Exibe a data atual com call_later()

Um exemplo de uma função de retorno mostrando a data atual a cada segundo. A função de retorno usa o método `loop.call_later()` para reagendar a si mesma depois de 5 segundos, e então para o laço de eventos:

```
import asyncio
import datetime

def display_date(end_time, loop):
    print(datetime.datetime.now())
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, display_date, end_time, loop)
    else:
        loop.stop()

loop = asyncio.get_event_loop()

# Schedule the first call to display_date()
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

Ver também:

Um exemplo similar a *data atual* criado com uma corrotina e a função `run()`.

Observa um descritor de arquivo por eventos de leitura

Aguarda até que um descritor de arquivo tenha recebido alguns dados usando o método `loop.add_reader()` e então fecha o laço de eventos:

```
import asyncio
from socket import socketpair

# Create a pair of connected file descriptors
rsock, wsock = socketpair()

loop = asyncio.get_event_loop()
```

(continua na próxima página)

(continuação da página anterior)

```
def reader():
    data = rsock.recv(100)
    print("Received:", data.decode())

    # We are done: unregister the file descriptor
    loop.remove_reader(rsock)

    # Stop the event loop
    loop.stop()

# Register the file descriptor for read event
loop.add_reader(rsock, reader)

# Simulate the reception of data from the network
loop.call_soon(wsock.send, 'abc'.encode())

try:
    # Run the event loop
    loop.run_forever()
finally:
    # We are done. Close sockets and the event loop.
    rsock.close()
    wsock.close()
    loop.close()
```

Ver também:

- Um *exemplo* similar usando transportes, protocolos, e o método `loop.create_connection()`.
- Outro *exemplo* similar usando a função de alto nível `asyncio.open_connection()` e streams.

Define tratadores de sinais para SIGINT e SIGTERM

(Este exemplo de `signals` apenas funciona no Unix.)

Registra tratadores para sinais `SIGINT` e `SIGTERM` usando o método `loop.add_signal_handler()`:

```
import asyncio
import functools
import os
import signal

def ask_exit(signame, loop):
    print("got signal %s: exit" % signame)
    loop.stop()

async def main():
    loop = asyncio.get_running_loop()

    for signame in {'SIGINT', 'SIGTERM'}:
        loop.add_signal_handler(
            getattr(signal, signame),
            functools.partial(ask_exit, signame, loop))

    await asyncio.sleep(3600)
```

(continua na próxima página)

(continuação da página anterior)

```
print("Event loop running for 1 hour, press Ctrl+C to interrupt.")
print(f"pid {os.getpid()}: send SIGINT or SIGTERM to exit.")

asyncio.run(main())
```

19.1.8 Futuros

Future objects are used to bridge **low-level callback-based code** with high-level async/await code.

Future Functions

asyncio.**isfuture**(*obj*)
Return `True` if *obj* is either of:

- an instance of *asyncio.Future*,
- an instance of *asyncio.Task*,
- a Future-like object with a `_asyncio_future_blocking` attribute.

Novo na versão 3.5.

asyncio.**ensure_future**(*obj*, *, *loop=None*)
Return:

- *obj* argument as is, if *obj* is a *Future*, a *Task*, or a Future-like object (*isfuture()* is used for the test.)
- a *Task* object wrapping *obj*, if *obj* is a coroutine (*iscoroutine()* is used for the test); in this case the coroutine will be scheduled by `ensure_future()`.
- a *Task* object that would await on *obj*, if *obj* is an awaitable (*inspect.isawaitable()* is used for the test.)

If *obj* is neither of the above a *TypeError* is raised.

Importante: See also the *create_task()* function which is the preferred way for creating new Tasks.

Alterado na versão 3.5.1: The function accepts any *awaitable* object.

asyncio.**wrap_future**(*future*, *, *loop=None*)
Wrap a *concurrent.futures.Future* object in a *asyncio.Future* object.

Future Object

class asyncio.**Future**(*, *loop=None*)
A Future represents an eventual result of an asynchronous operation. Not thread-safe.

Future is an *awaitable* object. Coroutines can await on Future objects until they either have a result or an exception set, or until they are cancelled.

Typically Futures are used to enable low-level callback-based code (e.g. in protocols implemented using asyncio *transports*) to interoperate with high-level async/await code.

The rule of thumb is to never expose Future objects in user-facing APIs, and the recommended way to create a Future object is to call *loop.create_future()*. This way alternative event loop implementations can inject their own optimized implementations of a Future object.

Alterado na versão 3.7: Adicionado suporte para o módulo *contextvars*.

result()
Return the result of the Future.

If the Future is *done* and has a result set by the *set_result()* method, the result value is returned.

If the Future is *done* and has an exception set by the *set_exception()* method, this method raises the exception.

Se o futuro foi *cancelled*, este método levanta uma exceção *CancelledError*.

If the Future's result isn't yet available, this method raises a *InvalidStateError* exception.

set_result(*result*)
Mark the Future as *done* and set its result.

Raises a *InvalidStateError* error if the Future is already *done*.

set_exception(*exception*)
Mark the Future as *done* and set an exception.

Raises a *InvalidStateError* error if the Future is already *done*.

done()
Return `True` if the Future is *done*.

A Future is *done* if it was *cancelled* or if it has a result or an exception set with *set_result()* or *set_exception()* calls.

cancelled()
Return `True` if the Future was *cancelled*.

The method is usually used to check if a Future is not *cancelled* before setting a result or an exception for it:

```
if not fut.cancelled():
    fut.set_result(42)
```

add_done_callback(*callback*, *, *context=None*)
Add a callback to be run when the Future is *done*.

The *callback* is called with the Future object as its only argument.

If the Future is already *done* when this method is called, the callback is scheduled with *loop.call_soon()*.

Um argumento opcional somente-nomeado *context* permite especificar um *contextvars.Context* customizado para executar na *função de retorno*. O contexto atual é usado quando nenhum *context* é fornecido.

functools.partial() can be used to pass parameters to the callback, e.g.:

```
# Call 'print("Future:", fut)' when "fut" is done.
fut.add_done_callback(
    functools.partial(print, "Future:"))
```

Alterado na versão 3.7: O parâmetro somente-nomeado *context* foi adicionado. Veja **PEP 567** para mais detalhes.

remove_done_callback(*callback*)
Remove *callback* da lista de funções de retorno.

Returns the number of callbacks removed, which is typically 1, unless a callback was added more than once.

cancel()
Cancel the Future and schedule callbacks.

If the Future is already *done* or *cancelled*, return `False`. Otherwise, change the Future's state to *cancelled*, schedule the callbacks, and return `True`.

exception()
Return the exception that was set on this Future.

The exception (or `None` if no exception was set) is returned only if the Future is *done*.

Se o futuro foi *cancelled*, este método levanta uma exceção *CancelledError*.

If the Future isn't *done* yet, this method raises an *InvalidStateError* exception.

get_loop()
Return the event loop the Future object is bound to.

Novo na versão 3.7.

This example creates a Future object, creates and schedules an asynchronous Task to set result for the Future, and waits until the Future has a result:

```
async def set_after(fut, delay, value):
    # Sleep for *delay* seconds.
    await asyncio.sleep(delay)

    # Set *value* as a result of *fut* Future.
    fut.set_result(value)

async def main():
    # Get the current event loop.
    loop = asyncio.get_running_loop()

    # Create a new Future object.
    fut = loop.create_future()

    # Run "set_after()" coroutine in a parallel Task.
    # We are using the low-level "loop.create_task()" API here because
    # we already have a reference to the event loop at hand.
    # Otherwise we could have just used "asyncio.create_task()".
    loop.create_task(
        set_after(fut, 1, '... world'))

    print('hello ...')

    # Wait until *fut* has a result (1 second) and print it.
    print(await fut)

asyncio.run(main())
```

Importante: The Future object was designed to mimic *concurrent.futures.Future*. Key differences include:

- unlike asyncio Futures, *concurrent.futures.Future* instances cannot be awaited.
- *asyncio.Future.result()* and *asyncio.Future.exception()* do not accept the *timeout* argument.
- *asyncio.Future.result()* and *asyncio.Future.exception()* raise an *InvalidStateError* exception when the Future is not *done*.

- Callbacks registered with *asyncio.Future.add_done_callback()* are not called immediately. They are scheduled with *loop.call_soon()* instead.
- asyncio Future is not compatible with the *concurrent.futures.wait()* and *concurrent.futures.as_completed()* functions.

19.1.9 Transports and Protocols

Prefácio

Transports and Protocols are used by the **low-level** event loop APIs such as *loop.create_connection()*. They use callback-based programming style and enable high-performance implementations of network or IPC protocols (e.g. HTTP).

Essentially, transports and protocols should only be used in libraries and frameworks and never in high-level asyncio applications.

This documentation page covers both *Transports* and *Protocols*.

Introdução

At the highest level, the transport is concerned with *how* bytes are transmitted, while the protocol determines *which* bytes to transmit (and to some extent when).

A different way of saying the same thing: a transport is an abstraction for a socket (or similar I/O endpoint) while a protocol is an abstraction for an application, from the transport's point of view.

Yet another view is the transport and protocol interfaces together define an abstract interface for using network I/O and interprocess I/O.

There is always a 1:1 relationship between transport and protocol objects: the protocol calls transport methods to send data, while the transport calls protocol methods to pass it data that has been received.

Most of connection oriented event loop methods (such as *loop.create_connection()*) usually accept a *protocol_factory* argument used to create a *Protocol* object for an accepted connection, represented by a *Transport* object. Such methods usually return a tuple of `(transport, protocol)`.

Conteúdo

Esta página de documentação contém as seguintes seções:

- The *Transports* section documents asyncio *BaseTransport*, *ReadTransport*, *WriteTransport*, *Transport*, *DatagramTransport*, and *SubprocessTransport* classes.
- The *Protocols* section documents asyncio *BaseProtocol*, *Protocol*, *BufferedProtocol*, *DatagramProtocol*, and *SubprocessProtocol* classes.
- The *Examples* section showcases how to work with transports, protocols, and low-level event loop APIs.

Transportes

Transports are classes provided by `asyncio` in order to abstract various kinds of communication channels.

Transport objects are always instantiated by an *asyncio event loop*.

asyncio implements transports for TCP, UDP, SSL, and subprocess pipes. The methods available on a transport depend on the transport's kind.

The transport classes are *not thread safe*.

Transports Hierarchy

class asyncio.**BaseTransport**
Base class for all transports. Contains methods that all asyncio transports share.

class asyncio.**WriteTransport**(*BaseTransport*)
A base transport for write-only connections.

Instances of the *WriteTransport* class are returned from the `loop.connect_write_pipe()` event loop method and are also used by subprocess-related methods like `loop.subprocess_exec()`.

class asyncio.**ReadTransport**(*BaseTransport*)
A base transport for read-only connections.

Instances of the *ReadTransport* class are returned from the `loop.connect_read_pipe()` event loop method and are also used by subprocess-related methods like `loop.subprocess_exec()`.

class asyncio.**Transport**(*WriteTransport*, *ReadTransport*)
Interface representing a bidirectional transport, such as a TCP connection.

The user does not instantiate a transport directly; they call a utility function, passing it a protocol factory and other information necessary to create the transport and protocol.

Instances of the *Transport* class are returned from or used by event loop methods like `loop.create_connection()`, `loop.create_unix_connection()`, `loop.create_server()`, `loop.sendfile()`, etc.

class asyncio.**DatagramTransport**(*BaseTransport*)
A transport for datagram (UDP) connections.

Instances of the *DatagramTransport* class are returned from the `loop.create_datagram_endpoint()` event loop method.

class asyncio.**SubprocessTransport**(*BaseTransport*)
An abstraction to represent a connection between a parent and its child OS process.

Instances of the *SubprocessTransport* class are returned from event loop methods `loop.subprocess_shell()` and `loop.subprocess_exec()`.

Base Transport

BaseTransport.**close**()
Fecha o transporte.

If the transport has a buffer for outgoing data, buffered data will be flushed asynchronously. No more data will be received. After all buffered data is flushed, the protocol's *protocol.connection_lost()* method will be called with *None* as its argument.

BaseTransport.**is_closing**()
Retorna `True` se o transporte estiver fechando ou estiver fechado.

BaseTransport.**get_extra_info**(*name*, *default=None*)
Return information about the transport or underlying resources it uses.

name is a string representing the piece of transport-specific information to get.

default is the value to return if the information is not available, or if the transport does not support querying it with the given third-party event loop implementation or on the current platform.

For example, the following code attempts to get the underlying socket object of the transport:

```
sock = transport.get_extra_info('socket')
if sock is not None:
    print(sock.getsockopt(...))
```

Categories of information that can be queried on some transports:

- socket:
  - `'peername'`: the remote address to which the socket is connected, result of *socket.socket.getpeername()* (`None` on error)
  - `'socket'`: *socket.socket* instance
  - `'sockname'`: the socket's own address, result of *socket.socket.getsockname()*
- SSL socket:
  - `'compression'`: the compression algorithm being used as a string, or `None` if the connection isn't compressed; result of *ssl.SSLSocket.compression()*
  - `'cipher'`: a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used; result of *ssl.SSLSocket.cipher()*
  - `'peercert'`: peer certificate; result of *ssl.SSLSocket.getpeercert()*
  - `'sslcontext'`: *ssl.SSLContext* instance
  - `'ssl_object'`: *ssl.SSLObject* or *ssl.SSLSocket* instance
- pipe:
  - `'pipe'`: pipe object
- subprocess:
  - `'subprocess'`: *subprocess.Popen* instance

BaseTransport.**set_protocol**(*protocol*)
Define um novo protocolo.

Switching protocol should only be done when both protocols are documented to support the switch.

BaseTransport.**get_protocol**()
Retorna o protocolo atual.

Read-only Transports

ReadTransport.**is_reading**()
Return `True` if the transport is receiving new data.

Novo na versão 3.7.

ReadTransport.**pause_reading**()
Pause the receiving end of the transport. No data will be passed to the protocol's *protocol.data_received()* method until *resume_reading()* is called.

Alterado na versão 3.7: The method is idempotent, i.e. it can be called when the transport is already paused or closed.

ReadTransport.**resume_reading**()
Resume the receiving end. The protocol's *protocol.data_received()* method will be called once again if some data is available for reading.

Alterado na versão 3.7: The method is idempotent, i.e. it can be called when the transport is already reading.

Write-only Transports

WriteTransport.**abort**()
Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol's *protocol.connection_lost()* method will eventually be called with *None* as its argument.

WriteTransport.**can_write_eof**()
Return *True* if the transport supports *write_eof()*, *False* if not.

WriteTransport.**get_write_buffer_size**()
Return the current size of the output buffer used by the transport.

WriteTransport.**get_write_buffer_limits**()
Get the *high* and *low* watermarks for write flow control. Return a tuple `(low, high)` where *low* and *high* are positive number of bytes.

Use *set_write_buffer_limits()* to set the limits.

Novo na versão 3.4.2.

WriteTransport.**set_write_buffer_limits**(*high=None*, *low=None*)
Set the *high* and *low* watermarks for write flow control.

These two values (measured in number of bytes) control when the protocol's *protocol.pause_writing()* and *protocol.resume_writing()* methods are called. If specified, the low watermark must be less than or equal to the high watermark. Neither *high* nor *low* can be negative.

pause_writing() is called when the buffer size becomes greater than or equal to the *high* value. If writing has been paused, *resume_writing()* is called when the buffer size becomes less than or equal to the *low* value.

The defaults are implementation-specific. If only the high watermark is given, the low watermark defaults to an implementation-specific value less than or equal to the high watermark. Setting *high* to zero forces *low* to zero as well, and causes *pause_writing()* to be called whenever the buffer becomes non-empty. Setting *low* to zero causes *resume_writing()* to be called only once the buffer is empty. Use of zero for either limit is generally sub-optimal as it reduces opportunities for doing I/O and computation concurrently.

Use *get_write_buffer_limits()* to get the limits.

WriteTransport.**write**(*data*)
Write some *data* bytes to the transport.

This method does not block; it buffers the data and arranges for it to be sent out asynchronously.

WriteTransport.**writelines**(*list_of_data*)
Write a list (or any iterable) of data bytes to the transport. This is functionally equivalent to calling *write()* on each element yielded by the iterable, but may be implemented more efficiently.

WriteTransport.**write_eof**()
Close the write end of the transport after flushing all buffered data. Data may still be received.

This method can raise *NotImplementedError* if the transport (e.g. SSL) doesn't support half-closed connections.

Transportes de datagrama

DatagramTransport.**sendto**(*data*, *addr=None*)
Send the *data* bytes to the remote peer given by *addr* (a transport-dependent target address). If *addr* is *None*, the data is sent to the target address given on transport creation.

This method does not block; it buffers the data and arranges for it to be sent out asynchronously.

DatagramTransport.**abort**()
Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol's *protocol.connection_lost()* method will eventually be called with *None* as its argument.

Transportes de Subprocesso

SubprocessTransport.**get_pid**()
Return the subprocess process id as an integer.

SubprocessTransport.**get_pipe_transport**(*fd*)
Return the transport for the communication pipe corresponding to the integer file descriptor *fd*:

- 0: readable streaming transport of the standard input (*stdin*), or *None* if the subprocess was not created with stdin=PIPE
- 1: writable streaming transport of the standard output (*stdout*), or *None* if the subprocess was not created with stdout=PIPE
- 2: writable streaming transport of the standard error (*stderr*), or *None* if the subprocess was not created with stderr=PIPE
- other *fd*: *None*

SubprocessTransport.**get_returncode**()
Return the subprocess return code as an integer or *None* if it hasn't returned, which is similar to the *subprocess.Popen.returncode* attribute.

SubprocessTransport.**kill**()
Mata o subprocesso.

On POSIX systems, the function sends SIGKILL to the subprocess. On Windows, this method is an alias for *terminate()*.

See also *subprocess.Popen.kill()*.

SubprocessTransport.**send_signal**(*signal*)
Send the *signal* number to the subprocess, as in *subprocess.Popen.send_signal()*.

SubprocessTransport.**terminate**()
Interrompe o subprocesso.

On POSIX systems, this method sends SIGTERM to the subprocess. On Windows, the Windows API function TerminateProcess() is called to stop the subprocess.

See also *subprocess.Popen.terminate()*.

SubprocessTransport.**close**()
Kill the subprocess by calling the *kill()* method.

If the subprocess hasn't returned yet, and close transports of *stdin*, *stdout*, and *stderr* pipes.

Protocolos

asyncio provides a set of abstract base classes that should be used to implement network protocols. Those classes are meant to be used together with *transports*.

Subclasses of abstract base protocol classes may implement some or all methods. All these methods are callbacks: they are called by transports on certain events, for example when some data is received. A base protocol method should be called by the corresponding transport.

Protocolos de Base

class asyncio.**BaseProtocol**
Base protocol with methods that all protocols share.

class asyncio.**Protocol**(*BaseProtocol*)
The base class for implementing streaming protocols (TCP, Unix sockets, etc).

class asyncio.**BufferedProtocol**(*BaseProtocol*)
A base class for implementing streaming protocols with manual control of the receive buffer.

class asyncio.**DatagramProtocol**(*BaseProtocol*)
The base class for implementing datagram (UDP) protocols.

class asyncio.**SubprocessProtocol**(*BaseProtocol*)
The base class for implementing protocols communicating with child processes (unidirectional pipes).

Base Protocol

All asyncio protocols can implement Base Protocol callbacks.

Connection Callbacks

Connection callbacks are called on all protocols, exactly once per a successful connection. All other protocol callbacks can only be called between those two methods.

`BaseProtocol.`**`connection_made`**(*transport*)
Chamado quando uma conexão é estabelecida.

The *transport* argument is the transport representing the connection. The protocol is responsible for storing the reference to its transport.

`BaseProtocol.`**`connection_lost`**(*exc*)
Chamado quanto a conexão é perdida ou fechada.

The argument is either an exception object or *None*. The latter means a regular EOF is received, or the connection was aborted or closed by this side of the connection.

Flow Control Callbacks

Flow control callbacks can be called by transports to pause or resume writing performed by the protocol.

See the documentation of the *set_write_buffer_limits()* method for more details.

`BaseProtocol.`**`pause_writing`**()
Called when the transport's buffer goes over the high watermark.

`BaseProtocol.`**`resume_writing`**()
Called when the transport's buffer drains below the low watermark.

If the buffer size equals the high watermark, *pause_writing()* is not called: the buffer size must go strictly over.

Conversely, *resume_writing()* is called when the buffer size is equal or lower than the low watermark. These end conditions are important to ensure that things go as expected when either mark is zero.

Streaming Protocols

Event methods, such as *loop.create_server()*, *loop.create_unix_server()*, *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_accepted_socket()*, *loop.connect_read_pipe()*, and *loop.connect_write_pipe()* accept factories that return streaming protocols.

`Protocol.`**`data_received`**(*data*)
Called when some data is received. *data* is a non-empty bytes object containing the incoming data.

Whether the data is buffered, chunked or reassembled depends on the transport. In general, you shouldn't rely on specific semantics and instead make your parsing generic and flexible. However, data is always received in the correct order.

The method can be called an arbitrary number of times while a connection is open.

However, *protocol.eof_received()* is called at most once. Once *eof_received()* is called, `data_received()` is not called anymore.

`Protocol.`**`eof_received`**()
Called when the other end signals it won't send any more data (for example by calling *transport.write_eof()*, if the other end also uses asyncio).

This method may return a false value (including `None`), in which case the transport will close itself. Conversely, if this method returns a true value, the protocol used determines whether to close the transport. Since the default implementation returns `None`, it implicitly closes the connection.

Some transports, including SSL, don't support half-closed connections, in which case returning true from this method will result in the connection being closed.

State machine:

```
start -> connection_made
    [-> data_received]*
    [-> eof_received]?
-> connection_lost -> end
```

Protocolos de Streaming Bufferizados

Novo na versão 3.7: **Important:** this has been added to asyncio in Python 3.7 *on a provisional basis*! This is as an experimental API that might be changed or removed completely in Python 3.8.

Buffered Protocols can be used with any event loop method that supports *Streaming Protocols*.

`BufferedProtocol` implementations allow explicit manual allocation and control of the receive buffer. Event loops can then use the buffer provided by the protocol to avoid unnecessary data copies. This can result in noticeable performance improvement for protocols that receive big amounts of data. Sophisticated protocol implementations can significantly reduce the number of buffer allocations.

The following callbacks are called on *`BufferedProtocol`* instances:

`BufferedProtocol.`**`get_buffer`**(*sizehint*)
Chamada para alocar um novo buffer para recebimento.

sizehint is the recommended minimum size for the returned buffer. It is acceptable to return smaller or larger buffers than what *sizehint* suggests. When set to -1, the buffer size can be arbitrary. It is an error to return a buffer with a zero size.

`get_buffer()` must return an object implementing the buffer protocol.

`BufferedProtocol.`**`buffer_updated`**(*nbytes*)
Chamado quando o buffer foi atualizado com os dados recebidos.

nbytes is the total number of bytes that were written to the buffer.

`BufferedProtocol.`**`eof_received`**()
See the documentation of the *`protocol.eof_received()`* method.

`get_buffer()` can be called an arbitrary number of times during a connection. However, *`protocol.eof_received()`* is called at most once and, if called, *`get_buffer()`* and *`buffer_updated()`* won't be called after it.

State machine:

```
start -> connection_made
    [-> get_buffer
        [-> buffer_updated]?
    ]*
    [-> eof_received]?
-> connection_lost -> end
```

Protocolos de Datagramas

Datagram Protocol instances should be constructed by protocol factories passed to the *loop.create_datagram_endpoint()* method.

`DatagramProtocol.`**`datagram_received`**(*data*, *addr*)
Called when a datagram is received. *data* is a bytes object containing the incoming data. *addr* is the address of the peer sending the data; the exact format depends on the transport.

`DatagramProtocol.`**`error_received`**(*exc*)
Called when a previous send or receive operation raises an *OSError*. *exc* is the *OSError* instance.

This method is called in rare conditions, when the transport (e.g. UDP) detects that a datagram could not be delivered to its recipient. In many conditions though, undeliverable datagrams will be silently dropped.

Nota: On BSD systems (macOS, FreeBSD, etc.) flow control is not supported for datagram protocols, because there is no reliable way to detect send failures caused by writing too many packets.

The socket always appears 'ready' and excess packets are dropped. An *OSError* with `errno` set to *errno.ENOBUFS* may or may not be raised; if it is raised, it will be reported to *DatagramProtocol.error_received()* but otherwise ignored.

Protocolos de Subprocesso

Datagram Protocol instances should be constructed by protocol factories passed to the *loop.subprocess_exec()* and *loop.subprocess_shell()* methods.

`SubprocessProtocol.`**`pipe_data_received`**(*fd*, *data*)
Called when the child process writes data into its stdout or stderr pipe.

fd is the integer file descriptor of the pipe.

data is a non-empty bytes object containing the received data.

`SubprocessProtocol.`**`pipe_connection_lost`**(*fd*, *exc*)
Chamado quando um dos encadeamentos comunicando com o processo filho é fechado.

fd is the integer file descriptor that was closed.

`SubprocessProtocol.`**`process_exited`**()
Chamado quando o processo filho encerrou.

Exemplos

TCP Echo Server

Create a TCP echo server using the *loop.create_server()* method, send back received data, and close the connection:

```
import asyncio


class EchoServerProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
```

(continua na próxima página)

(continuação da página anterior)

```
        print('Connection from {}'.format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
        self.transport.write(data)

        print('Close the client socket')
        self.transport.close()


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    server = await loop.create_server(
        lambda: EchoServerProtocol(),
        '127.0.0.1', 8888)

    async with server:
        await server.serve_forever()


asyncio.run(main())
```

Ver também:

The *TCP echo server using streams* example uses the high-level `asyncio.start_server()` function.

TCP Echo Client

A TCP echo client using the `loop.create_connection()` method, sends data, and waits until the connection is closed:

```
import asyncio


class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))

    def connection_lost(self, exc):
        print('The server closed the connection')
```

(continua na próxima página)

(continuação da página anterior)

```
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = 'Hello World!'

    transport, protocol = await loop.create_connection(
        lambda: EchoClientProtocol(message, on_con_lost),
        '127.0.0.1', 8888)

    # Wait until the protocol signals that the connection
    # is lost and close the transport.
    try:
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

Ver também:

The *TCP echo client using streams* example uses the high-level *asyncio.open_connection()* function.

UDP Echo Server

A UDP echo server, using the *loop.create_datagram_endpoint()* method, sends back received data:

```
import asyncio


class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)


async def main():
    print("Starting UDP server")

    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    # One protocol instance will be created to serve all
```

(continua na próxima página)

(continuação da página anterior)

```
    # client requests.
    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoServerProtocol(),
        local_addr=('127.0.0.1', 9999))

    try:
        await asyncio.sleep(3600)  # Serve for 1 hour.
    finally:
        transport.close()


asyncio.run(main())
```

UDP Echo Client

A UDP echo client, using the *loop.create_datagram_endpoint()* method, sends data and closes the transport when it receives the answer:

```
import asyncio


class EchoClientProtocol:
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost
        self.transport = None

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())

        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
        print("Connection closed")
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = "Hello World!"

    transport, protocol = await loop.create_datagram_endpoint(
```

(continua na próxima página)

(continuação da página anterior)

```
        lambda: EchoClientProtocol(message, on_con_lost),
        remote_addr=('127.0.0.1', 9999))

    try:
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

Connecting Existing Sockets

Wait until a socket receives data using the *loop.create_connection()* method with a protocol:

```
import asyncio
import socket


class MyProtocol(asyncio.Protocol):

    def __init__(self, on_con_lost):
        self.transport = None
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        self.transport = transport

    def data_received(self, data):
        print("Received:", data.decode())

        # We are done: close the transport;
        # connection_lost() will be called automatically.
        self.transport.close()

    def connection_lost(self, exc):
        # The socket has been closed
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()
    on_con_lost = loop.create_future()

    # Create a pair of connected sockets
    rsock, wsock = socket.socketpair()

    # Register the socket to wait for data.
    transport, protocol = await loop.create_connection(
        lambda: MyProtocol(on_con_lost), sock=rsock)

    # Simulate the reception of data from the network.
    loop.call_soon(wsock.send, 'abc'.encode())
```

(continua na próxima página)

(continuação da página anterior)

```
    try:
        await protocol.on_con_lost
    finally:
        transport.close()
        wsock.close()

asyncio.run(main())
```

Ver também:

The *watch a file descriptor for read events* example uses the low-level `loop.add_reader()` method to register an FD.

The *register an open socket to wait for data using streams* example uses high-level streams created by the `open_connection()` function in a coroutine.

loop.subprocess_exec() and SubprocessProtocol

An example of a subprocess protocol used to get the output of a subprocess and to wait for the subprocess exit.

The subprocess is created by th `loop.subprocess_exec()` method:

```
import asyncio
import sys

class DateProtocol(asyncio.SubprocessProtocol):
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()

    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exit_future.set_result(True)

async def get_date():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    code = 'import datetime; print(datetime.datetime.now())'
    exit_future = asyncio.Future(loop=loop)

    # Create the subprocess controlled by DateProtocol;
    # redirect the standard output into a pipe.
    transport, protocol = await loop.subprocess_exec(
        lambda: DateProtocol(exit_future),
        sys.executable, '-c', code,
        stdin=None, stderr=None)

    # Wait for the subprocess exit using the process_exited()
    # method of the protocol.
    await exit_future
```

(continua na próxima página)

(continuação da página anterior)

```
    # Close the stdout pipe.
    transport.close()

    # Read the output which was collected by the
    # pipe_data_received() method of the protocol.
    data = bytes(protocol.output)
    return data.decode('ascii').rstrip()

if sys.platform == "win32":
    asyncio.set_event_loop_policy(
        asyncio.WindowsProactorEventLoopPolicy())

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

See also the *same example* written using high-level APIs.

19.1.10 Políticas

Uma política de laço de eventos é um objeto global por processo que controla o gerenciamento do laço de eventos. Cada laço de eventos possui uma política padrão, que pode ser alterada e personalizada usando a API de política.

Uma política define a noção de *contexto* e gerencia um laço de eventos separado por contexto. A política padrão define *context* como a thread atual.

By using a custom event loop policy, the behavior of *get_event_loop()*, *set_event_loop()*, and *new_event_loop()* functions can be customized.

Policy objects should implement the APIs defined in the *AbstractEventLoopPolicy* abstract base class.

Getting and Setting the Policy

The following functions can be used to get and set the policy for the current process:

asyncio.**get_event_loop_policy**()
Retorna a política de todo o processo atual.

asyncio.**set_event_loop_policy**(*policy*)
Set the current process-wide policy to *policy*.

If *policy* is set to None, the default policy is restored.

Policy Objects

The abstract event loop policy base class is defined as follows:

class asyncio.**AbstractEventLoopPolicy**
An abstract base class for asyncio policies.

get_event_loop()
Pegar o evento de loop para o contexto atual.

Return an event loop object implementing the *AbstractEventLoop* interface.

This method should never return None.

Alterado na versão 3.6.

set_event_loop(*loop*)
Definir o evento de loop do contexto atual como *loop*.

new_event_loop()
Cria e retorna um novo objeto de laço de eventos.

This method should never return `None`.

get_child_watcher()
Get a child process watcher object.

Return a watcher object implementing the *AbstractChildWatcher* interface.

This function is Unix specific.

set_child_watcher(*watcher*)
Set the current child process watcher to *watcher*.

This function is Unix specific.

asyncio ships with the following built-in policies:

class asyncio.**DefaultEventLoopPolicy**
The default asyncio policy. Uses *SelectorEventLoop* on both Unix and Windows platforms.

There is no need to install the default policy manually. asyncio is configured to use the default policy automatically.

class asyncio.**WindowsProactorEventLoopPolicy**
An alternative event loop policy that uses the *ProactorEventLoop* event loop implementation.

Disponibilidade: Windows.

Monitores de processos

A process watcher allows customization of how an event loop monitors child processes on Unix. Specifically, the event loop needs to know when a child process has exited.

In asyncio, child processes are created with *create_subprocess_exec()* and *loop.subprocess_exec()* functions.

asyncio defines the *AbstractChildWatcher* abstract base class, which child watchers should implement, and has two different implementations: *SafeChildWatcher* (configured to be used by default) and *FastChildWatcher*.

Veja também a seção *Subprocesso e Threads*.

The following two functions can be used to customize the child process watcher implementation used by the asyncio event loop:

asyncio.**get_child_watcher**()
Return the current child watcher for the current policy.

asyncio.**set_child_watcher**(*watcher*)
Set the current child watcher to *watcher* for the current policy. *watcher* must implement methods defined in the *AbstractChildWatcher* base class.

Nota: Third-party event loops implementations might not support custom child watchers. For such event loops, using *set_child_watcher()* might be prohibited or have no effect.

---

class asyncio.**AbstractChildWatcher**

add_child_handler(*pid*, *callback*, **args*)
Register a new child handler.

Arrange for `callback(pid, returncode, *args)` to be called when a process with PID equal to *pid* terminates. Specifying another callback for the same process replaces the previous handler.

The *callback* callable must be thread-safe.

remove_child_handler(*pid*)
Remove o manipulador para processo com PID igual a *pid*.

The function returns `True` if the handler was successfully removed, `False` if there was nothing to remove.

attach_loop(*loop*)
Attach the watcher to an event loop.

If the watcher was previously attached to an event loop, then it is first detached before attaching to the new loop.

Note: loop may be `None`.

close()
Close the watcher.

This method has to be called to ensure that underlying resources are cleaned-up.

class asyncio.**SafeChildWatcher**
This implementation avoids disrupting other code spawning processes by polling every process explicitly on a `SIGCHLD` signal.

This is a safe solution but it has a significant overhead when handling a big number of processes (*O(n)* each time a `SIGCHLD` is received).

asyncio uses this safe implementation by default.

class asyncio.**FastChildWatcher**
This implementation reaps every terminated processes by calling `os.waitpid(-1)` directly, possibly breaking other code spawning processes and waiting for their termination.

There is no noticeable overhead when handling a big number of children (*O(1)* each time a child terminates).

Custom Policies

To implement a new event loop policy, it is recommended to subclass *`DefaultEventLoopPolicy`* and override the methods for which custom behavior is wanted, e.g.:

```
class MyEventLoopPolicy(asyncio.DefaultEventLoopPolicy):

    def get_event_loop(self):
        """Get the event loop.

        This may be None or an instance of EventLoop.
        """
        loop = super().get_event_loop()
        # Do something with loop ...
        return loop

asyncio.set_event_loop_policy(MyEventLoopPolicy())
```

19.1.11 Suporte a plataformas

The *asyncio* module is designed to be portable, but some platforms have subtle differences and limitations due to the platforms' underlying architecture and capabilities.

Todas as plataformas

- *loop.add_reader()* and *loop.add_writer()* cannot be used to monitor file I/O.

Windows

All event loops on Windows do not support the following methods:

- *loop.create_unix_connection()* and *loop.create_unix_server()* are not supported. The *socket.AF_UNIX* socket family is specific to Unix.
- *loop.add_signal_handler()* and *loop.remove_signal_handler()* are not supported.

SelectorEventLoop has the following limitations:

- *SelectSelector* is used to wait on socket events: it supports sockets and is limited to 512 sockets.
- *loop.add_reader()* and *loop.add_writer()* only accept socket handles (e.g. pipe file descriptors are not supported).
- Pipes are not supported, so the *loop.connect_read_pipe()* and *loop.connect_write_pipe()* methods are not implemented.
- *Subprocesses* are not supported, i.e. *loop.subprocess_exec()* and *loop.subprocess_shell()* methods are not implemented.

ProactorEventLoop has the following limitations:

- The *loop.create_datagram_endpoint()* method is not supported.
- The *loop.add_reader()* and *loop.add_writer()* methods are not supported.

The resolution of the monotonic clock on Windows is usually around 15.6 msec. The best resolution is 0.5 msec. The resolution depends on the hardware (availability of HPET) and on the Windows configuration.

Suporte para subprocesso no Windows

SelectorEventLoop on Windows does not support subproceses. On Windows, *ProactorEventLoop* should be used instead:

```
import asyncio

asyncio.set_event_loop_policy(
    asyncio.WindowsProactorEventLoopPolicy())

asyncio.run(your_code())
```

The *policy.set_child_watcher()* function is also not supported, as *ProactorEventLoop* has a different mechanism to watch child processes.

macOS

Modern macOS versions are fully supported.

macOS <= 10.8

On macOS 10.6, 10.7 and 10.8, the default event loop uses *selectors.KqueueSelector*, which does not support character devices on these versions. The *SelectorEventLoop* can be manually configured to use *SelectSelector* or *PollSelector* to support character devices on these older versions of macOS. Example:

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

19.1.12 Índice da API de alto nível

Esta página lista todas as APIs asyncio de alto nível habilitadas por async/await.

Tarefas

Utilitários para executar programas asyncio, criar Tasks, e esperar por múltiplas coisas com tempos limites.

run()	Cria um loop de evento, roda uma corrotina, fecha o loop.
create_task()	Inicia uma Task de asyncio.
await *sleep()*	Dorme for um número de segundos.
await *gather()*	Agenda e espera pelas coisas concorrentemente.
await *wait_for()*	Executar com um tempo limite.
await *shield()*	Proteger contra cancelamento.
await *wait()*	Monitorar para for conclusão.
current_task()	Retorna para a Task atual.
all_tasks()	Retorna todas as tasks para um loop de evento.
Task	Objeto Task.
run_coroutine_threadsafe()	Agenda uma corrotina a partir de outra thread do sistema operacional.
for in *as_completed()*	Monitora a conclusão com um loop `for`.

Exemplos

- *Usando asyncio.gather() para executar coisas em paralelo*.
- *Usando asyncio.wait_for() para forçar um tempo limite de execução*.
- *Cancelamento*.
- *Usando asyncio.sleep()*.
- Veja também a *página principal de documentação sobre Tasks*.

Filas

Filas devem ser usadas para distribuir trabalho entre múltiplos asyncio Tasks, implementar pools de conexão, e padrões pub/sub.

Queue	Uma fila FIFO - Primeiro que entra, é o primeiro que sai.
PriorityQueue	Uma fila de prioridade.
LifoQueue	Uma fila LIFO - Último que entra, é o primeiro que sai.

Exemplos

- *Usando asyncio.Queue para distribuir cargas de trabalho entre diversas Tasks*.
- Veja também a *Página de documentação da classe Queue*.

Subprocesso

Utilitários para iniciar subprocessos e executar comandos no console.

await *create_subprocess_exec()*	Cria um subprocesso.
await *create_subprocess_shell()*	Executa um comando no console.

Exemplos

- *Executando um comando no console*.
- Veja também a *documentação de subprocessos de APIs*.

Fluxos

APIs de alto nível para trabalhar com entrada e saída de rede.

await *open_connection()*	Estabelece uma conexão TCP.
await *open_unix_connection()*	Estabelece uma conexão com soquete Unix.
await *start_server()*	Inicia um servidor TCP.
await *start_unix_server()*	Inicia um servidor com soquete Unix.
StreamReader	Objeto async/await de alto nível para receber dados de rede.
StreamWriter	Objeto async/await de alto nível para enviar dados pela rede.

Exemplos

- *Exemplo de cliente TCP*.
- Veja também a *documentação de fluxos APIs*.

Sincronização

Primitivas de sincronização similares a threads, que podem ser usadas em tarefas.

`Lock`	Um bloqueio mutex.
`Event`	Um objeto de evento.
`Condition`	Um objeto de condição.
`Semaphore`	Um semáforo.
`BoundedSemaphore`	Um semáforo limitado.

Exemplos

- *Usando asyncio.Event*.
- Veja também a documentação das *primitivas de sincronização de asyncio*.

Exceções

`asyncio.TimeoutError`	Levantado ao atingir o tempo limite de funções como `wait_for()`. Tenha em mente que `asyncio.TimeoutError` **não é relacionado** com a exceção `TimeoutError` embutida.
`asyncio.CancelledError`	Levantado quanto a Task é cancelada. Veja também `Task.cancel()`.

Exemplos

- *Manipulando CancelledError para executar código no cancelamento de uma requisição*.
- Veja também a lista completa de *exceções específicas de asyncio*.

19.1.13 Índice de APIs de baixo nível

Esta página lista todas as APIs de baixo nível do asyncio.

Obtendo o Loop de Eventos

asyncio.get_running_loop()	A função **preferida** para obter o laço de eventos em execução.
asyncio.get_event_loop()	Obtém uma instância de laço de eventos (atual ou através da política).
asyncio.set_event_loop()	Define o laço de eventos como atual através da política atual.
asyncio.new_event_loop()	Cria um novo laço de eventos.

Exemplos

- *Usando asyncio.get_running_loop()*.

Métodos do laço de eventos

Veja também a seção principal da documentação sobre os *métodos de laço de eventos*.

Ciclo de vida

loop.run_until_complete()	Executa um Future/Task/aguardável até que esteja completo.
loop.run_forever()	Executa o laço de eventos para sempre.
loop.stop()	Para o laço de eventos.
loop.close()	Fecha o loop de eventos.
loop.is_running()	Retorna `True` se o laço de eventos estiver rodando.
loop.is_closed()	Retorna `True` se o laço de eventos estiver fechado.
`await` *loop.shutdown_asyncgens()*	Fecha geradores assíncronos.

Depuração

loop.set_debug()	Habilita ou desabilita o modo de debug.
loop.get_debug()	Obtém o modo de debug atual.

Agendando funções de retorno (callbacks)

loop.call_soon()	Invoca uma função de retorno brevemente.
loop.call_soon_threadsafe()	Uma variante segura para thread de *loop.call_soon()*.
loop.call_later()	Invoca uma função de retorno *após* o tempo especificado.
loop.call_at()	Invoca uma função de retorno *no* instante especificado.

Grupo de Thread/Processo

`await loop.run_in_executor()`	Executa uma função vinculada à CPU ou outra que seja bloqueante em um executor `concurrent.futures`.
`loop.set_default_executor()`	Define o executor padrão para `loop.run_in_executor()`.

Tasks e Futures

`loop.create_future()`	Cria um objeto `Future`.
`loop.create_task()`	Agenda corrotina como uma `Task`.
`loop.set_task_factory()`	Define uma factory usada por `loop.create_task()` para criar `Tasks`.
`loop.get_task_factory()`	Obtém o factory `loop.create_task()` usado para criar `Tasks`.

DNS

`await loop.getaddrinfo()`	Versão assíncrona de `socket.getaddrinfo()`.
`await loop.getnameinfo()`	Versão assíncrona de `socket.getnameinfo()`.

Rede e IPC

`await loop.create_connection()`	Abre uma conexão TCP.
`await loop.create_server()`	Cria um servidor TCP.
`await loop.create_unix_connection()`	Abre uma conexão soquete Unix.
`await loop.create_unix_server()`	Cria um servidor soquete Unix.
`await loop.connect_accepted_socket()`	Envolve um `socket` em um par `(transport, protocol)`.
`await loop.create_datagram_endpoint()`	Abre uma conexão por datagrama (UDP).
`await loop.sendfile()`	Envia um arquivo por meio de um transporte.
`await loop.start_tls()`	Atualiza uma conexão existente para TLS.
`await loop.connect_read_pipe()`	Envolve a leitura final de um encadeamento em um par `(transport, protocol)`.
`await loop.connect_write_pipe()`	Envolve a escrita final de um encadeamento em um par `(transport, protocol)`.

Sockets

`await loop.sock_recv()`	Recebe dados do *socket*.
`await loop.sock_recv_into()`	Recebe dados do *socket* em um buffer.
`await loop.sock_sendall()`	Envia dados para o *socket*.
`await loop.sock_connect()`	Conecta ao *socket*.
`await loop.sock_accept()`	Aceita uma conexão do *socket*.
`await loop.sock_sendfile()`	Envia um arquivo usando o *socket*.
loop.add_reader()	Começa a observar um descritor de arquivo, aguardando por disponibilidade de leitura.
loop.remove_reader()	Interrompe o monitoramento de um descritor de arquivo, que aguarda disponibilidade de leitura.
loop.add_writer()	Começa a observar um descritor de arquivo, aguardando por disponibilidade para escrita.
loop.remove_writer()	Interrompe o monitoramento de um descritor de arquivo, que aguarda disponibilidade para escrita.

Sinais Unix

loop.add_signal_handler()	Adiciona um tratador para um *signal*.
loop.remove_signal_handler()	Remove um tratador para um *signal*.

Subprocesso

loop.subprocess_exec()	Inicia um subprocesso.
loop.subprocess_shell()	Inicia um subprocesso a partir de um comando shell.

Tratamento de erros

loop.call_exception_handler()	Chama o tratamento de exceção.
loop.set_exception_handler()	Define um novo tratador de exceção.
loop.get_exception_handler()	Obtém o tratador de exceção atual.
loop.default_exception_handler()	A implementação padrão do tratador de exceção.

Exemplos

- *Usando asyncio.get_event_loop() e loop.run_forever()*.
- *Usando loop.call_later()*.
- Usando `loop.create_connection()` para implementar *um cliente-eco*.
- Usando `loop.create_connection()` para *conectar a um soquete*.
- *Usando add_reader() para monitorar um descritor de arquivo para eventos de leitura*.
- *Usando loop.add_signal_handler()*.
- *Usando loop.subprocess_exec()*.

Transportes

Todos os transportes implementam os seguintes métodos:

transport.close()	Fecha o transporte.
transport.is_closing()	Retorna `True` se o transporte estiver fechando ou estiver fechado.
transport.get_extra_info()	Solicita informação a respeito do transporte.
transport.set_protocol()	Define um novo protocolo.
transport.get_protocol()	Retorna o protocolo atual.

Transportes que podem receber dados (TCP e conexões Unix, encadeamentos, etc). Retornado a partir de métodos como *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_read_pipe()*, etc:

Realiza leitura de Transportes

transport.is_reading()	Retorna `True` se o transporte estiver recebendo.
transport.pause_reading()	Pausa o recebimento.
transport.resume_reading()	Continua o recebimento.

Transportes que podem enviar dados (TCP e conexões Unix, encadeamentos, etc). Retornado a partir de métodos como *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_write_pipe()*, etc:

Realiza escrita de Transportes

transport.write()	Escreve dados para o transporte.
transport.writelines()	Escreve buffers para o transporte.
transport.can_write_eof()	Retorna *True* se o transporte suporta o envio de EOF.
transport.write_eof()	Fecha e envia EOF após descarregar dados que estavam no buffer.
transport.abort()	Feche o transporte imediatamente.
transport.get_write_buffer_size()	Retorna marcas d'agua alta e baixa para controle do fluxo de escrita.
transport.set_write_buffer_limits()	Define novas marcas d'agua alta e baixa para controle do fluxo de escrita.

Transporte retornado por *loop.create_datagram_endpoint()*:

Transportes de datagrama

transport.sendto()	Envia dados para o par remoto.
transport.abort()	Feche o transporte imediatamente.

Abstração de transporte de baixo nível sobre subprocessos. Retornado por *loop.subprocess_exec()* e *loop.subprocess_shell()*:

Transportes de Subprocesso

transport.get_pid()	Retorna o process id do subprocesso.
transport.get_pipe_transport()	Retorna o transporte para o encadeamento de comunicação requisitada (*stdin*, *stdout*, ou *stderr*).
transport.get_returncode()	Retorna o código de retorno do subprocesso.
transport.kill()	Mata o subprocesso.
transport.send_signal()	Envia um sinal para o subprocesso.
transport.terminate()	Interrompe o subprocesso.
transport.close()	Mata o subprocesso e fecha todos os encadeamentos.

Protocolos

Classes de protocolos podem implementar os seguintes **métodos de função de retorno**:

callback *connection_made()*	Chamado quando uma conexão é estabelecida.
callback *connection_lost()*	Chamado quanto a conexão é perdida ou fechada.
callback *pause_writing()*	Chamado quando o buffer de transporte ultrapassa a marca de nível alto d'agua.
callback *resume_writing()*	Chamado quando o buffer de transporte drena abaixo da marca de nível baixo d'agua.

Protocolos de Streaming (TCP, Soquetes Unix, Encadeamentos)

callback *data_received()*	Chamado quando algum dado é recebido.
callback *eof_received()*	Chamado quando um EOF é recebido.

Protocolos de Streaming Bufferizados

callback *get_buffer()*	Chamada para alocar um novo buffer para recebimento.
callback *buffer_updated()*	Chamado quando o buffer foi atualizado com os dados recebidos.
callback *eof_received()*	Chamado quando um EOF é recebido.

Protocolos de Datagramas

`callback` :meth:‘datagram_received() ‘	Chamado quando um datagrama é recebido.
`callback` :meth:‘error_received() ‘	Chamado quando uma operação de envio ou recebimento anterior levanta um *`OSError`*.

Protocolos de Subprocesso

`callback` *`pipe_data_received()`*	Chamado quando o processo filho escreve dados no seu encadeamento *stdout* ou *stderr*.
`callback` *`pipe_connection_lost()`*	Chamado quando um dos encadeamentos comunicando com o processo filho é fechado.
`callback` *`process_exited()`*	Chamado quando o processo filho encerrou.

Políticas de laço de eventos

Políticas é um mecanismo de baixo nível para alterar o comportamento de funções, similar a *`asyncio.get_event_loop()`*. Veja também a *seção principal de políticas* para mais detalhes.

Acessando Políticas

`asyncio.get_event_loop_policy()`	Retorna a política de todo o processo atual.
`asyncio.set_event_loop_policy()`	Define uma nova política para todo o processo.
`AbstractEventLoopPolicy`	Classe base para objetos de política.

19.1.14 Desenvolvendo com asyncio

Asynchronous programming is different from classic “sequential” programming.

This page lists common mistakes and traps and explains how to avoid them.

Modo de Depuração

By default asyncio runs in production mode. In order to ease the development asyncio has a *debug mode*.

There are several ways to enable asyncio debug mode:

- Setting the `PYTHONASYNCIODEBUG` environment variable to `1`.
- Using the `-X dev` Python command line option.
- Passando `debug=True` para *`asyncio.run()`*.
- Chamando *`loop.set_debug()`*.

Além de habilitar o modo de depuração, considere também:

- setting the log level of the *asyncio logger* to `logging.DEBUG`, for example the following snippet of code can be run at startup of the application:

```
logging.basicConfig(level=logging.DEBUG)
```

- configuring the *warnings* module to display *ResourceWarning* warnings. One way of doing that is by using the `-W default` command line option.

Quando o modo de depuração está habilitado:

- asyncio checks for *coroutines that were not awaited* and logs them; this mitigates the "forgotten await" pitfall.
- Many non-threadsafe asyncio APIs (such as *loop.call_soon()* and *loop.call_at()* methods) raise an exception if they are called from a wrong thread.
- O tempo de execução de um seletor I/O é registrado se demorar muito para executar a operação I/O.
- Callbacks taking longer than 100ms are logged. The `loop.slow_callback_duration` attribute can be used to set the minimum execution duration in seconds that is considered "slow".

Concorrência e Múltiplas Threads

An event loop runs in a thread (typically the main thread) and executes all callbacks and Tasks in its thread. While a Task is running in the event loop, no other Tasks can run in the same thread. When a Task executes an `await` expression, the running Task gets suspended, and the event loop executes the next Task.

To schedule a callback from a different OS thread, the *loop.call_soon_threadsafe()* method should be used. Example:

```
loop.call_soon_threadsafe(callback, *args)
```

Almost all asyncio objects are not thread safe, which is typically not a problem unless there is code that works with them from outside of a Task or a callback. If there's a need for such code to call a low-level asyncio API, the *loop.call_soon_threadsafe()* method should be used, e.g.:

```
loop.call_soon_threadsafe(fut.cancel)
```

To schedule a coroutine object from a different OS thread, the *run_coroutine_threadsafe()* function should be used. It returns a *concurrent.futures.Future* to access the result:

```
async def coro_func():
    return await asyncio.sleep(1, 42)

# Later in another OS thread:

future = asyncio.run_coroutine_threadsafe(coro_func(), loop)
# Wait for the result:
result = future.result()
```

Para tratar sinais e executar subprocessos, o laço de eventos precisa executar na thread principal.

The *loop.run_in_executor()* method can be used with a *concurrent.futures.ThreadPoolExecutor* to execute blocking code in a different OS thread without blocking the OS thread that the event loop runs in.

Executando código bloqueante

Blocking (CPU-bound) code should not be called directly. For example, if a function performs a CPU-intensive calculation for 1 second, all concurrent asyncio Tasks and IO operations would be delayed by 1 second.

An executor can be used to run a task in a different thread or even in a different process to avoid blocking the OS thread with the event loop. See the *loop.run_in_executor()* method for more details.

Gerando logs

asyncio usa o módulo *logging* e todo registro é feito via o registrador `"asyncio"`.

O nível de log padrão é `logging.INFO`, mas pode ser facilmente ajustado:

```
logging.getLogger("asyncio").setLevel(logging.WARNING)
```

Detect never-awaited coroutines

When a coroutine function is called, but not awaited (e.g. `coro()` instead of `await coro()`) or the coroutine is not scheduled with *asyncio.create_task()*, asyncio will emit a *RuntimeWarning*:

```
import asyncio

async def test():
    print("never scheduled")

async def main():
    test()

asyncio.run(main())
```

Saída:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
  test()
```

Output in debug mode:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
Coroutine created at (most recent call last)
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

  < .. >

  File "../t.py", line 7, in main
    test()
  test()
```

The usual fix is to either await the coroutine or call the *asyncio.create_task()* function:

```
async def main():
    await test()
```

Detect never-retrieved exceptions

If a *Future.set_exception()* is called but the Future object is never awaited on, the exception would never be propagated to the user code. In this case, asyncio would emit a log message when the Future object is garbage collected.

Exemplo de uma exceção não tratada:

```
import asyncio

async def bug():
    raise Exception("not consumed")

async def main():
    asyncio.create_task(bug())

asyncio.run(main())
```

Saída:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
  exception=Exception('not consumed')>

Traceback (most recent call last):
  File "test.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

Enable the debug mode to get the traceback where the task was created:

```
asyncio.run(main(), debug=True)
```

Output in debug mode:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
    exception=Exception('not consumed') created at asyncio/tasks.py:321>

source_traceback: Object created at (most recent call last):
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

< .. >

Traceback (most recent call last):
  File "../t.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

19.2 socket — Interface de rede de baixo nível

Código Fonte: Lib/socket.py

Este módulo provê acesso à interface de *soquete* do BSD. Está disponível em todos os sistemas modernos Unix, Windows, MacOS e provavelmente outras plataformas.

Nota: Algum comportamento pode depender da plataforma, pois as chamadas são feitas para as APIs de soquete do sistema operacional.

A interface Python é uma transliteração direta da chamada de sistema e interface de biblioteca de soquetes do Unix para o estilo orientado a objetos do Python: a função *socket()* retorna um *objeto socket* cujos métodos implementam as diversas chamadas de sistema de soquetes. Os tipos de parâmetro são consideravelmente de nível mais alto que os da interface em C: assim como as operações `read()` e `write()` em arquivos Python, a alocação de buffer em operações de recebimento é automática, e o comprimento do buffer é implícito em operações de envio.

Ver também:

Módulo *socketserver* Classes that simplify writing network servers.

Module *ssl* A TLS/SSL wrapper for socket objects.

19.2.1 Famílias de soquete

Dependendo do sistema e das opções de construção, várias famílias de soquetes são suportadas por este módulo.

The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created. Socket addresses are represented as follows:

- The address of an *AF_UNIX* socket bound to a file system node is represented as a string, using the file system encoding and the `'surrogateescape'` error handler (see **PEP 383**). An address in Linux's abstract namespace is returned as a *bytes-like object* with an initial null byte; note that sockets in this namespace can communicate with normal file system sockets, so programs intended to run on Linux may need to deal with both types of address. A string or bytes-like object can be used for either type of address when passing it as an argument.

  Alterado na versão 3.3: Previously, *AF_UNIX* socket paths were assumed to use UTF-8 encoding.

  Alterado na versão 3.5: Writable *bytes-like object* is now accepted.

- A pair `(host, port)` is used for the *AF_INET* address family, where *host* is a string representing either a hostname in Internet domain notation like `'daring.cwi.nl'` or an IPv4 address like `'100.50.200.5'`, and *port* is an integer.

  - For IPv4 addresses, two special forms are accepted instead of a host address: `''` represents `INADDR_ANY`, which is used to bind to all interfaces, and the string `'<broadcast>'` represents `INADDR_BROADCAST`. This behavior is not compatible with IPv6, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.

- For *AF_INET6* address family, a four-tuple `(host, port, flowinfo, scopeid)` is used, where *flowinfo* and *scopeid* represent the `sin6_flowinfo` and `sin6_scope_id` members in `struct sockaddr_in6` in C. For *socket* module methods, *flowinfo* and *scopeid* can be omitted just for backward compatibility. Note, however, omission of *scopeid* can cause problems in manipulating scoped IPv6 addresses.

  Alterado na versão 3.7: For multicast addresses (with *scopeid* meaningful) *address* may not contain `%scope` (or `zone id`) part. This information is superfluous and may be safely omitted (recommended).

- AF_NETLINK sockets are represented as pairs (pid, groups).
- Linux-only support for TIPC is available using the AF_TIPC address family. TIPC is an open, non-IP based networked protocol designed for use in clustered computer environments. Addresses are represented by a tuple, and the fields depend on the address type. The general tuple form is (addr_type, v1, v2, v3 [, scope]), where:
  - *addr_type* is one of TIPC_ADDR_NAMESEQ, TIPC_ADDR_NAME, or TIPC_ADDR_ID.
  - *scope* is one of TIPC_ZONE_SCOPE, TIPC_CLUSTER_SCOPE, and TIPC_NODE_SCOPE.
  - If *addr_type* is TIPC_ADDR_NAME, then *v1* is the server type, *v2* is the port identifier, and *v3* should be 0.

    If *addr_type* is TIPC_ADDR_NAMESEQ, then *v1* is the server type, *v2* is the lower port number, and *v3* is the upper port number.

    If *addr_type* is TIPC_ADDR_ID, then *v1* is the node, *v2* is the reference, and *v3* should be set to 0.
- A tuple (interface,) is used for the *AF_CAN* address family, where *interface* is a string representing a network interface name like 'can0'. The network interface name '' can be used to receive packets from all network interfaces of this family.
  - *CAN_ISOTP* protocol require a tuple (interface, rx_addr, tx_addr) where both additional parameters are unsigned long integer that represent a CAN identifier (standard or extended).
- A string or a tuple (id, unit) is used for the SYSPROTO_CONTROL protocol of the PF_SYSTEM family. The string is the name of a kernel control using a dynamically-assigned ID. The tuple can be used if ID and unit number of the kernel control are known or if a registered ID is used.

  Novo na versão 3.3.
- AF_BLUETOOTH supports the following protocols and address formats:
  - BTPROTO_L2CAP accepts (bdaddr, psm) where bdaddr is the Bluetooth address as a string and psm is an integer.
  - BTPROTO_RFCOMM accepts (bdaddr, channel) where bdaddr is the Bluetooth address as a string and channel is an integer.
  - BTPROTO_HCI accepts (device_id,) where device_id is either an integer or a string with the Bluetooth address of the interface. (This depends on your OS; NetBSD and DragonFlyBSD expect a Bluetooth address while everything else expects an integer.)

    Alterado na versão 3.2: NetBSD and DragonFlyBSD support added.
  - BTPROTO_SCO accepts bdaddr where bdaddr is a *bytes* object containing the Bluetooth address in a string format. (ex. b'12:23:34:45:56:67') This protocol is not supported under FreeBSD.
- *AF_ALG* is a Linux-only socket based interface to Kernel cryptography. An algorithm socket is configured with a tuple of two to four elements (type, name [, feat [, mask]]), where:
  - *type* is the algorithm type as string, e.g. aead, hash, skcipher or rng.
  - *name* is the algorithm name and operation mode as string, e.g. sha256, hmac(sha256), cbc(aes) or drbg_nopr_ctr_aes256.
  - *feat* and *mask* are unsigned 32bit integers.

  Availability: Linux 2.6.38, some algorithm types require more recent Kernels.

  Novo na versão 3.6.
- *AF_VSOCK* allows communication between virtual machines and their hosts. The sockets are represented as a (CID, port) tuple where the context ID or CID and port are integers.

  Availability: Linux >= 4.8 QEMU >= 2.8 ESX >= 4.0 ESX Workstation >= 6.5.

Novo na versão 3.7.

- *AF_PACKET* is a low-level interface directly to network devices. The packets are represented by the tuple `(ifname, proto[, pkttype[, hatype[, addr]]])` where:
  - *ifname* - String specifying the device name.
  - *proto* - An in network-byte-order integer specifying the Ethernet protocol number.
  - *pkttype* - Optional integer specifying the packet type:
    - `PACKET_HOST` (the default) - Packet addressed to the local host.
    - `PACKET_BROADCAST` - Physical-layer broadcast packet.
    - `PACKET_MULTIHOST` - Packet sent to a physical-layer multicast address.
    - `PACKET_OTHERHOST` - Packet to some other host that has been caught by a device driver in promiscuous mode.
    - `PACKET_OUTGOING` - Packet originating from the local host that is looped back to a packet socket.
  - *hatype* - Optional integer specifying the ARP hardware address type.
  - *addr* - Optional bytes-like object specifying the hardware physical address, whose interpretation depends on the device.

If you use a hostname in the *host* portion of IPv4/v6 socket address, the program may show a nondeterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in *host* portion.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised; starting from Python 3.3, errors related to socket or address semantics raise *OSError* or one of its subclasses (they used to raise *socket.error*).

Non-blocking mode is supported through *setblocking()*. A generalization of this based on timeouts is supported through *settimeout()*.

19.2.2 Conteúdo do módulo

The module *socket* exports the following elements.

Exceções

exception socket.**error**
A deprecated alias of *OSError*.

Alterado na versão 3.3: Following **PEP 3151**, this class was made an alias of *OSError*.

exception socket.**herror**
A subclass of *OSError*, this exception is raised for address-related errors, i.e. for functions that use *h_errno* in the POSIX C API, including *gethostbyname_ex()* and *gethostbyaddr()*. The accompanying value is a pair `(h_errno, string)` representing an error returned by a library call. *h_errno* is a numeric value, while *string* represents the description of *h_errno*, as returned by the `hstrerror()` C function.

Alterado na versão 3.3: This class was made a subclass of *OSError*.

exception socket.**gaierror**
A subclass of *OSError*, this exception is raised for address-related errors by *getaddrinfo()* and *getnameinfo()*. The accompanying value is a pair `(error, string)` representing an error returned by

a library call. *string* represents the description of *error*, as returned by the `gai_strerror()` C function. The numeric *error* value will match one of the `EAI_*` constants defined in this module.

Alterado na versão 3.3: This class was made a subclass of *OSError*.

exception socket.**timeout**
A subclass of *OSError*, this exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to *settimeout()* (or implicitly through *setdefaulttimeout()*). The accompanying value is a string whose value is currently always "timed out".

Alterado na versão 3.3: This class was made a subclass of *OSError*.

Constantes

The AF_* and SOCK_* constants are now `AddressFamily` and `SocketKind` *IntEnum* collections.

Novo na versão 3.4.

socket.**AF_UNIX**
socket.**AF_INET**
socket.**AF_INET6**
These constants represent the address (and protocol) families, used for the first argument to *socket()*. If the *AF_UNIX* constant is not defined then this protocol is unsupported. More constants may be available depending on the system.

socket.**SOCK_STREAM**
socket.**SOCK_DGRAM**
socket.**SOCK_RAW**
socket.**SOCK_RDM**
socket.**SOCK_SEQPACKET**
These constants represent the socket types, used for the second argument to *socket()*. More constants may be available depending on the system. (Only *SOCK_STREAM* and *SOCK_DGRAM* appear to be generally useful.)

socket.**SOCK_CLOEXEC**
socket.**SOCK_NONBLOCK**
These two constants, if defined, can be combined with the socket types and allow you to set some flags atomically (thus avoiding possible race conditions and the need for separate calls).

Ver também:

Secure File Descriptor Handling for a more thorough explanation.

Disponibilidade: Linux >= 2.6.27.

Novo na versão 3.2.

SO_*
socket.**SOMAXCONN**
MSG_*
SOL_*
SCM_*
IPPROTO_*
IPPORT_*
INADDR_*
IP_*
IPV6_*
EAI_*
AI_*
NI_*

TCP_*
Many constants of these forms, documented in the Unix documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the `setsockopt()` and `getsockopt()` methods of socket objects. In most cases, only those symbols that are defined in the Unix header files are defined; for a few symbols, default values are provided.

Alterado na versão 3.6: `SO_DOMAIN`, `SO_PROTOCOL`, `SO_PEERSEC`, `SO_PASSSEC`, `TCP_USER_TIMEOUT`, `TCP_CONGESTION` were added.

Alterado na versão 3.6.5: On Windows, `TCP_FASTOPEN`, `TCP_KEEPCNT` appear if run-time Windows supports.

Alterado na versão 3.7: `TCP_NOTSENT_LOWAT` was added.

On Windows, `TCP_KEEPIDLE`, `TCP_KEEPINTVL` appear if run-time Windows supports.

`socket.`**AF_CAN**
`socket.`**PF_CAN**
SOL_CAN_*
CAN_*
Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Disponibilidade: Linux >= 2.6.25.

Novo na versão 3.3.

`socket.`**CAN_BCM**
CAN_BCM_*
CAN_BCM, in the CAN protocol family, is the broadcast manager (BCM) protocol. Broadcast manager constants, documented in the Linux documentation, are also defined in the socket module.

Disponibilidade: Linux >= 2.6.25.

Novo na versão 3.4.

`socket.`**CAN_RAW_FD_FRAMES**
Enables CAN FD support in a CAN_RAW socket. This is disabled by default. This allows your application to send both CAN and CAN FD frames; however, you must accept both CAN and CAN FD frames when reading from the socket.

This constant is documented in the Linux documentation.

Disponibilidade: Linux >= 3.6.

Novo na versão 3.5.

`socket.`**CAN_ISOTP**
CAN_ISOTP, in the CAN protocol family, is the ISO-TP (ISO 15765-2) protocol. ISO-TP constants, documented in the Linux documentation.

Disponibilidade: Linux >= 2.6.25.

Novo na versão 3.7.

`socket.`**AF_PACKET**
`socket.`**PF_PACKET**
PACKET_*
Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Disponibilidade: Linux >= 2.2.

`socket.`**AF_RDS**
`socket.`**PF_RDS**
`socket.`**SOL_RDS**

RDS_*
Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Disponibilidade: Linux >= 2.6.30.

Novo na versão 3.3.

socket.**SIO_RCVALL**
socket.**SIO_KEEPALIVE_VALS**
socket.**SIO_LOOPBACK_FAST_PATH**
RCVALL_*
Constants for Windows' WSAIoctl(). The constants are used as arguments to the *ioctl()* method of socket objects.

Alterado na versão 3.6: SIO_LOOPBACK_FAST_PATH was added.

TIPC_*
TIPC related constants, matching the ones exported by the C socket API. See the TIPC documentation for more information.

socket.**AF_ALG**
socket.**SOL_ALG**
ALG_*
Constants for Linux Kernel cryptography.

Disponibilidade: Linux >= 2.6.38.

Novo na versão 3.6.

socket.**AF_VSOCK**
socket.**IOCTL_VM_SOCKETS_GET_LOCAL_CID**
VMADDR*
SO_VM*
Constants for Linux host/guest communication.

Disponibilidade: Linux >= 4.8.

Novo na versão 3.7.

socket.**AF_LINK**
Availability: BSD, OSX.

Novo na versão 3.4.

socket.**has_ipv6**
This constant contains a boolean value which indicates if IPv6 is supported on this platform.

socket.**BDADDR_ANY**
socket.**BDADDR_LOCAL**
These are string constants containing Bluetooth addresses with special meanings. For example, *BDADDR_ANY* can be used to indicate any address when specifying the binding socket with BTPROTO_RFCOMM.

socket.**HCI_FILTER**
socket.**HCI_TIME_STAMP**
socket.**HCI_DATA_DIR**
For use with BTPROTO_HCI. *HCI_FILTER* is not available for NetBSD or DragonFlyBSD. *HCI_TIME_STAMP* and *HCI_DATA_DIR* are not available for FreeBSD, NetBSD, or DragonFlyBSD.

Funções

Criação de sockets

The following functions all create *socket objects*.

socket.**socket**(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*)

Create a new socket using the given address family, socket type and protocol number. The address family should be *AF_INET* (the default), *AF_INET6*, *AF_UNIX*, *AF_CAN*, *AF_PACKET*, or *AF_RDS*. The socket type should be *SOCK_STREAM* (the default), *SOCK_DGRAM*, *SOCK_RAW* or perhaps one of the other SOCK_ constants. The protocol number is usually zero and may be omitted or in the case where the address family is *AF_CAN* the protocol should be one of CAN_RAW, *CAN_BCM* or *CAN_ISOTP*.

If *fileno* is specified, the values for *family*, *type*, and *proto* are auto-detected from the specified file descriptor. Auto-detection can be overruled by calling the function with explicit *family*, *type*, or *proto* arguments. This only affects how Python represents e.g. the return value of *socket.getpeername()* but not the actual OS resource. Unlike *socket.fromfd()*, *fileno* will return the same socket and not a duplicate. This may help close a detached socket using *socket.close()*.

The newly created socket is *non-inheritable*.

Alterado na versão 3.3: The AF_CAN family was added. The AF_RDS family was added.

Alterado na versão 3.4: The CAN_BCM protocol was added.

Alterado na versão 3.4: The returned socket is now non-inheritable.

Alterado na versão 3.7: The CAN_ISOTP protocol was added.

Alterado na versão 3.7: When *SOCK_NONBLOCK* or *SOCK_CLOEXEC* bit flags are applied to *type* they are cleared, and *socket.type* will not reflect them. They are still passed to the underlying system *socket()* call. Therefore,

```
sock = socket.socket(
    socket.AF_INET,
    socket.SOCK_STREAM | socket.SOCK_NONBLOCK)
```

will still create a non-blocking socket on OSes that support SOCK_NONBLOCK, but `sock.type` will be set to `socket.SOCK_STREAM`.

socket.**socketpair**([*family*[, *type*[, *proto*]]])

Build a pair of connected socket objects using the given address family, socket type, and protocol number. Address family, socket type, and protocol number are as for the *socket()* function above. The default family is *AF_UNIX* if defined on the platform; otherwise, the default is *AF_INET*.

The newly created sockets are *non-inheritable*.

Alterado na versão 3.2: The returned socket objects now support the whole socket API, rather than a subset.

Alterado na versão 3.4: The returned sockets are now non-inheritable.

Alterado na versão 3.5: Windows support added.

socket.**create_connection**(*address*[, *timeout*[, *source_address*]])

Connect to a TCP service listening on the Internet *address* (a 2-tuple `(host, port)`), and return the socket object. This is a higher-level function than *socket.connect()*: if *host* is a non-numeric hostname, it will try to resolve it for both *AF_INET* and *AF_INET6*, and then try to connect to all possible addresses in turn until a connection succeeds. This makes it easy to write clients that are compatible to both IPv4 and IPv6.

Passing the optional *timeout* parameter will set the timeout on the socket instance before attempting to connect. If no *timeout* is supplied, the global default timeout setting returned by *getdefaulttimeout()* is used.

If supplied, *source_address* must be a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting. If host or port are '' or 0 respectively the OS default behavior will be used.

Alterado na versão 3.2: *source_address* foi adicionado.

`socket.`**`fromfd`**(*fd*, *family*, *type*, *proto=0*)
Duplicate the file descriptor *fd* (an integer as returned by a file object's `fileno()` method) and build a socket object from the result. Address family, socket type and protocol number are as for the *socket()* function above. The file descriptor should refer to a socket, but this is not checked — subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (such as a server started by the Unix inet daemon). The socket is assumed to be in blocking mode.

The newly created socket is *non-inheritable*.

Alterado na versão 3.4: The returned socket is now non-inheritable.

`socket.`**`fromshare`**(*data*)
Instantiate a socket from data obtained from the *socket.share()* method. The socket is assumed to be in blocking mode.

Disponibilidade: Windows.

Novo na versão 3.3.

`socket.`**`SocketType`**
This is a Python type object that represents the socket object type. It is the same as `type(socket(...))`.

Outras funções

The *socket* module also offers various network-related services:

`socket.`**`close`**(*fd*)
Close a socket file descriptor. This is like *os.close()*, but for sockets. On some platforms (most noticeable Windows) *os.close()* does not work for socket file descriptors.

Novo na versão 3.7.

`socket.`**`getaddrinfo`**(*host*, *port*, *family=0*, *type=0*, *proto=0*, *flags=0*)
Translate the *host*/*port* argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. *host* is a domain name, a string representation of an IPv4/v6 address or `None`. *port* is a string service name such as `'http'`, a numeric port number or `None`. By passing `None` as the value of *host* and *port*, you can pass `NULL` to the underlying C API.

The *family*, *type* and *proto* arguments can be optionally specified in order to narrow the list of addresses returned. Passing zero as a value for each of these arguments selects the full range of results. The *flags* argument can be one or several of the `AI_*` constants, and will influence how results are computed and returned. For example, `AI_NUMERICHOST` will disable domain name resolution and will raise an error if *host* is a domain name.

The function returns a list of 5-tuples with the following structure:

```
(family, type, proto, canonname, sockaddr)
```

In these tuples, *family*, *type*, *proto* are all integers and are meant to be passed to the *socket()* function. *canonname* will be a string representing the canonical name of the *host* if `AI_CANONNAME` is part of the *flags* argument; else *canonname* will be empty. *sockaddr* is a tuple describing a socket address, whose format depends on the returned *family* (a `(address, port)` 2-tuple for *AF_INET*, a `(address, port, flow info, scope id)` 4-tuple for *AF_INET6*), and is meant to be passed to the *socket.connect()* method.

The following example fetches address information for a hypothetical TCP connection to `example.org` on port 80 (results may differ on your system if IPv6 isn't enabled):

```
>>> socket.getaddrinfo("example.org", 80, proto=socket.IPPROTO_TCP)
[(<AddressFamily.AF_INET6: 10>, <SocketType.SOCK_STREAM: 1>,
 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (<AddressFamily.AF_INET: 2>, <SocketType.SOCK_STREAM: 1>,
 6, '', ('93.184.216.34', 80))]
```

Alterado na versão 3.2: parameters can now be passed using keyword arguments.

Alterado na versão 3.7: for IPv6 multicast addresses, string representing an address will not contain `%scope` part.

`socket.`**`getfqdn`**`(`[*name*]`)`

Return a fully qualified domain name for *name*. If *name* is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by *gethostbyaddr()* is checked, followed by aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available, the hostname as returned by *gethostname()* is returned.

`socket.`**`gethostbyname`**`(`*hostname*`)`

Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as `'100.50.200.5'`. If the host name is an IPv4 address itself it is returned unchanged. See *gethostbyname_ex()* for a more complete interface. *gethostbyname()* does not support IPv6 name resolution, and *getaddrinfo()* should be used instead for IPv4/v6 dual stack support.

`socket.`**`gethostbyname_ex`**`(`*hostname*`)`

Translate a host name to IPv4 address format, extended interface. Return a triple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the primary host name responding to the given *ip_address*, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4 addresses for the same interface on the same host (often but not always a single address). *gethostbyname_ex()* does not support IPv6 name resolution, and *getaddrinfo()* should be used instead for IPv4/v6 dual stack support.

`socket.`**`gethostname`**`()`

Return a string containing the hostname of the machine where the Python interpreter is currently executing.

Note: *gethostname()* doesn't always return the fully qualified domain name; use *getfqdn()* for that.

`socket.`**`gethostbyaddr`**`(`*ip_address*`)`

Return a triple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the primary host name responding to the given *ip_address*, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4/v6 addresses for the same interface on the same host (most likely containing only a single address). To find the fully qualified domain name, use the function *getfqdn()*. *gethostbyaddr()* supports both IPv4 and IPv6.

`socket.`**`getnameinfo`**`(`*sockaddr*, *flags*`)`

Translate a socket address *sockaddr* into a 2-tuple `(host, port)`. Depending on the settings of *flags*, the result can contain a fully-qualified domain name or numeric address representation in *host*. Similarly, *port* can contain a string port name or a numeric port number.

For IPv6 addresses, `%scope` is appended to the host part if *sockaddr* contains meaningful *scopeid*. Usually this happens for multicast addresses.

`socket.`**`getprotobyname`**`(`*protocolname*`)`

Translate an Internet protocol name (for example, `'icmp'`) to a constant suitable for passing as the (optional) third argument to the *socket()* function. This is usually only needed for sockets opened in "raw" mode (*SOCK_RAW*); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.

`socket.`**`getservbyname`**`(`*servicename*[, *protocolname*]`)`

Translate an Internet service name and protocol name to a port number for that service. The optional protocol name, if given, should be `'tcp'` or `'udp'`, otherwise any protocol will match.

`socket.`**`getservbyport`**`(`*port*[, *protocolname*]`)`

Translate an Internet port number and protocol name to a service name for that service. The optional protocol

name, if given, should be `'tcp'` or `'udp'`, otherwise any protocol will match.

`socket.`**`ntohl`**(*x*)
Convert 32-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

`socket.`**`ntohs`**(*x*)
Convert 16-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

Obsoleto desde a versão 3.7: In case *x* does not fit in 16-bit unsigned integer, but does fit in a positive C int, it is silently truncated to 16-bit unsigned integer. This silent truncation feature is deprecated, and will raise an exception in future versions of Python.

`socket.`**`htonl`**(*x*)
Convert 32-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

`socket.`**`htons`**(*x*)
Convert 16-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

Obsoleto desde a versão 3.7: In case *x* does not fit in 16-bit unsigned integer, but does fit in a positive C int, it is silently truncated to 16-bit unsigned integer. This silent truncation feature is deprecated, and will raise an exception in future versions of Python.

`socket.`**`inet_aton`**(*ip_string*)
Convert an IPv4 address from dotted-quad string format (for example, '123.45.67.89') to 32-bit packed binary format, as a bytes object four characters in length. This is useful when conversing with a program that uses the standard C library and needs objects of type `struct in_addr`, which is the C type for the 32-bit packed binary this function returns.

`inet_aton()` also accepts strings with less than three dots; see the Unix manual page *`inet(3)`* for details.

If the IPv4 address string passed to this function is invalid, *`OSError`* will be raised. Note that exactly what is valid depends on the underlying C implementation of `inet_aton()`.

`inet_aton()` does not support IPv6, and *`inet_pton()`* should be used instead for IPv4/v6 dual stack support.

`socket.`**`inet_ntoa`**(*packed_ip*)
Convert a 32-bit packed IPv4 address (a *bytes-like object* four bytes in length) to its standard dotted-quad string representation (for example, '123.45.67.89'). This is useful when conversing with a program that uses the standard C library and needs objects of type `struct in_addr`, which is the C type for the 32-bit packed binary data this function takes as an argument.

If the byte sequence passed to this function is not exactly 4 bytes in length, *`OSError`* will be raised. *`inet_ntoa()`* does not support IPv6, and *`inet_ntop()`* should be used instead for IPv4/v6 dual stack support.

Alterado na versão 3.5: Writable *bytes-like object* is now accepted.

`socket.`**`inet_pton`**(*address_family*, *ip_string*)
Convert an IP address from its family-specific string format to a packed, binary format. *`inet_pton()`* is useful when a library or network protocol calls for an object of type `struct in_addr` (similar to *`inet_aton()`*) or `struct in6_addr`.

Supported values for *address_family* are currently *`AF_INET`* and *`AF_INET6`*. If the IP address string *ip_string* is invalid, *`OSError`* will be raised. Note that exactly what is valid depends on both the value of *address_family* and the underlying implementation of `inet_pton()`.

Availability: Unix (maybe not all platforms), Windows.

Alterado na versão 3.4: Suporte para Windows adicionado.

socket.**inet_ntop**(*address_family*, *packed_ip*)
Convert a packed IP address (a *bytes-like object* of some number of bytes) to its standard, family-specific string representation (for example, `'7.10.0.5'` or `'5aef:2b::8'`). *inet_ntop()* is useful when a library or network protocol returns an object of type `struct in_addr` (similar to *inet_ntoa()*) or `struct in6_addr`.

Supported values for *address_family* are currently *AF_INET* and *AF_INET6*. If the bytes object *packed_ip* is not the correct length for the specified address family, *ValueError* will be raised. *OSError* is raised for errors from the call to *inet_ntop()*.

Availability: Unix (maybe not all platforms), Windows.

Alterado na versão 3.4: Suporte para Windows adicionado.

Alterado na versão 3.5: Writable *bytes-like object* is now accepted.

socket.**CMSG_LEN**(*length*)
Return the total length, without trailing padding, of an ancillary data item with associated data of the given *length*. This value can often be used as the buffer size for *recvmsg()* to receive a single item of ancillary data, but **RFC 3542** requires portable applications to use *CMSG_SPACE()* and thus include space for padding, even when the item will be the last in the buffer. Raises *OverflowError* if *length* is outside the permissible range of values.

Availability: most Unix platforms, possibly others.

Novo na versão 3.3.

socket.**CMSG_SPACE**(*length*)
Return the buffer size needed for *recvmsg()* to receive an ancillary data item with associated data of the given *length*, along with any trailing padding. The buffer space needed to receive multiple items is the sum of the *CMSG_SPACE()* values for their associated data lengths. Raises *OverflowError* if *length* is outside the permissible range of values.

Note that some systems might support ancillary data without providing this function. Also note that setting the buffer size using the results of this function may not precisely limit the amount of ancillary data that can be received, since additional data may be able to fit into the padding area.

Availability: most Unix platforms, possibly others.

Novo na versão 3.3.

socket.**getdefaulttimeout**()
Return the default timeout in seconds (float) for new socket objects. A value of `None` indicates that new socket objects have no timeout. When the socket module is first imported, the default is `None`.

socket.**setdefaulttimeout**(*timeout*)
Set the default timeout in seconds (float) for new socket objects. When the socket module is first imported, the default is `None`. See *settimeout()* for possible values and their respective meanings.

socket.**sethostname**(*name*)
Set the machine's hostname to *name*. This will raise an *OSError* if you don't have enough rights.

Disponibilidade: Unix.

Novo na versão 3.3.

socket.**if_nameindex**()
Return a list of network interface information (index int, name string) tuples. *OSError* if the system call fails.

Disponibilidade: Unix.

Novo na versão 3.3.

socket.**if_nametoindex**(*if_name*)
Return a network interface index number corresponding to an interface name. *OSError* if no interface with the given name exists.

Disponibilidade: Unix.

Novo na versão 3.3.

socket.**if_indextoname**(*if_index*)
Return a network interface name corresponding to an interface index number. *OSError* if no interface with the given index exists.

Disponibilidade: Unix.

Novo na versão 3.3.

19.2.3 Socket Objects

Socket objects have the following methods. Except for *makefile()*, these correspond to Unix system calls applicable to sockets.

Alterado na versão 3.2: Support for the *context manager* protocol was added. Exiting the context manager is equivalent to calling *close()*.

socket.**accept**()
Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair `(conn, address)` where *conn* is a *new* socket object usable to send and receive data on the connection, and *address* is the address bound to the socket on the other end of the connection.

The newly created socket is *non-inheritable*.

Alterado na versão 3.4: The socket is now non-inheritable.

Alterado na versão 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.**bind**(*address*)
Bind the socket to *address*. The socket must not already be bound. (The format of *address* depends on the address family — see above.)

socket.**close**()
Mark the socket closed. The underlying system resource (e.g. a file descriptor) is also closed when all file objects from *makefile()* are closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed).

Sockets are automatically closed when they are garbage-collected, but it is recommended to *close()* them explicitly, or to use a `with` statement around them.

Alterado na versão 3.6: *OSError* is now raised if an error occurs when the underlying `close()` call is made.

Nota: *close()* releases the resource associated with a connection but does not necessarily close the connection immediately. If you want to close the connection in a timely fashion, call *shutdown()* before *close()*.

---

socket.**connect**(*address*)
Connect to a remote socket at *address*. (The format of *address* depends on the address family — see above.)

If the connection is interrupted by a signal, the method waits until the connection completes, or raise a *socket.timeout* on timeout, if the signal handler doesn't raise an exception and the socket is blocking or has a timeout.

For non-blocking sockets, the method raises an *InterruptedError* exception if the connection is interrupted by a signal (or the exception raised by the signal handler).

Alterado na versão 3.5: The method now waits until the connection completes instead of raising an *InterruptedError* exception if the connection is interrupted by a signal, the signal handler doesn't raise an exception and the socket is blocking or has a timeout (see the **PEP 475** for the rationale).

socket.**connect_ex**(*address*)
Like `connect(address)`, but return an error indicator instead of raising an exception for errors returned by the C-level `connect()` call (other problems, such as "host not found," can still raise exceptions). The error indicator is `0` if the operation succeeded, otherwise the value of the `errno` variable. This is useful to support, for example, asynchronous connects.

socket.**detach**()
Put the socket object into closed state without actually closing the underlying file descriptor. The file descriptor is returned, and can be reused for other purposes.

Novo na versão 3.2.

socket.**dup**()
Duplicate the socket.

The newly created socket is *non-inheritable*.

Alterado na versão 3.4: The socket is now non-inheritable.

socket.**fileno**()
Return the socket's file descriptor (a small integer), or -1 on failure. This is useful with *select.select()*.

Under Windows the small integer returned by this method cannot be used where a file descriptor can be used (such as *os.fdopen()*). Unix does not have this limitation.

socket.**get_inheritable**()
Get the *inheritable flag* of the socket's file descriptor or socket's handle: `True` if the socket can be inherited in child processes, `False` if it cannot.

Novo na versão 3.4.

socket.**getpeername**()
Return the remote address to which the socket is connected. This is useful to find out the port number of a remote IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.) On some systems this function is not supported.

socket.**getsockname**()
Return the socket's own address. This is useful to find out the port number of an IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.)

socket.**getsockopt**(*level*, *optname*[, *buflen*])
Return the value of the given socket option (see the Unix man page *getsockopt(2)*). The needed symbolic constants (`SO_*` etc.) are defined in this module. If *buflen* is absent, an integer option is assumed and its integer value is returned by the function. If *buflen* is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a bytes object. It is up to the caller to decode the contents of the buffer (see the optional built-in module *struct* for a way to decode C structures encoded as byte strings).

socket.**getblocking**()
Return `True` if socket is in blocking mode, `False` if in non-blocking.

This is equivalent to checking `socket.gettimeout() == 0`.

Novo na versão 3.7.

socket.**gettimeout**()
Return the timeout in seconds (float) associated with socket operations, or None if no timeout is set. This reflects the last call to *setblocking()* or *settimeout()*.

socket.**ioctl**(*control*, *option*)

Platform Windows

The *ioctl()* method is a limited interface to the WSAIoctl system interface. Please refer to the Win32 documentation for more information.

On other platforms, the generic *fcntl.fcntl()* and *fcntl.ioctl()* functions may be used; they accept a socket object as their first argument.

Currently only the following control codes are supported: SIO_RCVALL, SIO_KEEPALIVE_VALS, and SIO_LOOPBACK_FAST_PATH.

Alterado na versão 3.6: SIO_LOOPBACK_FAST_PATH was added.

socket.**listen**([*backlog*])
Enable a server to accept connections. If *backlog* is specified, it must be at least 0 (if it is lower, it is set to 0); it specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.

Alterado na versão 3.5: The *backlog* parameter is now optional.

socket.**makefile**(*mode='r'*, *buffering=None*, ***, *encoding=None*, *errors=None*, *newline=None*)
Return a *file object* associated with the socket. The exact returned type depends on the arguments given to *makefile()*. These arguments are interpreted the same way as by the built-in *open()* function, except the only supported *mode* values are 'r' (default), 'w' and 'b'.

The socket must be in blocking mode; it can have a timeout, but the file object's internal buffer may end up in an inconsistent state if a timeout occurs.

Closing the file object returned by *makefile()* won't close the original socket unless all other file objects have been closed and *socket.close()* has been called on the socket object.

Nota: On Windows, the file-like object created by *makefile()* cannot be used where a file object with a file descriptor is expected, such as the stream arguments of *subprocess.Popen()*.

---

socket.**recv**(*bufsize*[, *flags*])
Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by *bufsize*. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero.

Nota: For best match with hardware and network realities, the value of *bufsize* should be a relatively small power of 2, for example, 4096.

Alterado na versão 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.**recvfrom**(*bufsize*[, *flags*])
Receive data from the socket. The return value is a pair (bytes, address) where *bytes* is a bytes object representing the data received and *address* is the address of the socket sending the data. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family — see above.)

Alterado na versão 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

Alterado na versão 3.7: For multicast IPv6 address, first item of *address* does not contain `%scope` part anymore. In order to get full IPv6 address use *getnameinfo()*.

`socket.`**recvmsg**(*bufsize*[, *ancbufsize*[, *flags*]])

Receive normal data (up to *bufsize* bytes) and ancillary data from the socket. The *ancbufsize* argument sets the size in bytes of the internal buffer used to receive the ancillary data; it defaults to 0, meaning that no ancillary data will be received. Appropriate buffer sizes for ancillary data can be calculated using *CMSG_SPACE()* or *CMSG_LEN()*, and items which do not fit into the buffer might be truncated or discarded. The *flags* argument defaults to 0 and has the same meaning as for *recv()*.

The return value is a 4-tuple: `(data, ancdata, msg_flags, address)`. The *data* item is a *bytes* object holding the non-ancillary data received. The *ancdata* item is a list of zero or more tuples `(cmsg_level, cmsg_type, cmsg_data)` representing the ancillary data (control messages) received: *cmsg_level* and *cmsg_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg_data* is a *bytes* object holding the associated data. The *msg_flags* item is the bitwise OR of various flags indicating conditions on the received message; see your system documentation for details. If the receiving socket is unconnected, *address* is the address of the sending socket, if available; otherwise, its value is unspecified.

On some systems, *sendmsg()* and *recvmsg()* can be used to pass file descriptors between processes over an *AF_UNIX* socket. When this facility is used (it is often restricted to *SOCK_STREAM* sockets), *recvmsg()* will return, in its ancillary data, items of the form `(socket.SOL_SOCKET, socket.SCM_RIGHTS, fds)`, where *fds* is a *bytes* object representing the new file descriptors as a binary array of the native C `int` type. If *recvmsg()* raises an exception after the system call returns, it will first attempt to close any file descriptors received via this mechanism.

Some systems do not indicate the truncated length of ancillary data items which have been only partially received. If an item appears to extend beyond the end of the buffer, *recvmsg()* will issue a *RuntimeWarning*, and will return the part of it which is inside the buffer provided it has not been truncated before the start of its associated data.

On systems which support the `SCM_RIGHTS` mechanism, the following function will receive up to *maxfds* file descriptors, returning the message data and a list containing the descriptors (while ignoring unexpected conditions such as unrelated control messages being received). See also *sendmsg()*.

```
import socket, array

def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")   # Array of ints
    msg, ancdata, flags, addr = sock.recvmsg(msglen, socket.CMSG_LEN(maxfds * fds.
↪itemsize))
    for cmsg_level, cmsg_type, cmsg_data in ancdata:
        if cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS:
            # Append data, ignoring any truncated integers at the end.
            fds.frombytes(cmsg_data[:len(cmsg_data) - (len(cmsg_data) % fds.
↪itemsize)])
    return msg, list(fds)
```

Availability: most Unix platforms, possibly others.

Novo na versão 3.3.

Alterado na versão 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.**recvmsg_into**(*buffers*[, *ancbufsize*[, *flags*]])

Receive normal data and ancillary data from the socket, behaving as *recvmsg()* would, but scatter the non-ancillary data into a series of buffers instead of returning a new bytes object. The *buffers* argument must be an iterable of objects that export writable buffers (e.g. *bytearray* objects); these will be filled with successive chunks of the non-ancillary data until it has all been written or there are no more buffers. The operating system may set a limit (*sysconf()* value SC_IOV_MAX) on the number of buffers that can be used. The *ancbufsize* and *flags* arguments have the same meaning as for *recvmsg()*.

The return value is a 4-tuple: (nbytes, ancdata, msg_flags, address), where *nbytes* is the total number of bytes of non-ancillary data written into the buffers, and *ancdata*, *msg_flags* and *address* are the same as for *recvmsg()*.

Exemplo:

```
>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'----')
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'--------------')
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'little lamb---')]
```

Availability: most Unix platforms, possibly others.

Novo na versão 3.3.

socket.**recvfrom_into**(*buffer*[, *nbytes*[, *flags*]])

Receive data from the socket, writing it into *buffer* instead of creating a new bytestring. The return value is a pair (nbytes, address) where *nbytes* is the number of bytes received and *address* is the address of the socket sending the data. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family — see above.)

socket.**recv_into**(*buffer*[, *nbytes*[, *flags*]])

Receive up to *nbytes* bytes from the socket, storing the data into a buffer rather than creating a new bytestring. If *nbytes* is not specified (or 0), receive up to the size available in the given buffer. Returns the number of bytes received. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero.

socket.**send**(*bytes*[, *flags*])

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for *recv()* above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data. For further information on this topic, consult the socket-howto.

Alterado na versão 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an *InterruptedError* exception (see **PEP 475** for the rationale).

socket.**sendall**(*bytes*[, *flags*])

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for *recv()* above. Unlike *send()*, this method continues to send data from *bytes* until either all data has been sent or an error occurs. None is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

Alterado na versão 3.5: The socket timeout is no more reset each time data is sent successfully. The socket timeout is now the maximum total duration to send all data.

Alterado na versão 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** for the rationale).

`socket.`**`sendto`**(*bytes*, *address*)
`socket.`**`sendto`**(*bytes*, *flags*, *address*)
Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*. The optional *flags* argument has the same meaning as for `recv()` above. Return the number of bytes sent. (The format of *address* depends on the address family — see above.)

Alterado na versão 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** for the rationale).

`socket.`**`sendmsg`**(*buffers*[, *ancdata*[, *flags*[, *address*]]])
Send normal and ancillary data to the socket, gathering the non-ancillary data from a series of buffers and concatenating it into a single message. The *buffers* argument specifies the non-ancillary data as an iterable of *bytes-like objects* (e.g. `bytes` objects); the operating system may set a limit (`sysconf()` value `SC_IOV_MAX`) on the number of buffers that can be used. The *ancdata* argument specifies the ancillary data (control messages) as an iterable of zero or more tuples `(cmsg_level, cmsg_type, cmsg_data)`, where *cmsg_level* and *cmsg_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg_data* is a bytes-like object holding the associated data. Note that some systems (in particular, systems without `CMSG_SPACE()`) might support sending only one control message per call. The *flags* argument defaults to 0 and has the same meaning as for `send()`. If *address* is supplied and not `None`, it sets a destination address for the message. The return value is the number of bytes of non-ancillary data sent.

The following function sends the list of file descriptors *fds* over an `AF_UNIX` socket, on systems which support the `SCM_RIGHTS` mechanism. See also `recvmsg()`.

```
import socket, array

def send_fds(sock, msg, fds):
    return sock.sendmsg([msg], [(socket.SOL_SOCKET, socket.SCM_RIGHTS, array.
↪array("i", fds))])
```

Availability: most Unix platforms, possibly others.

Novo na versão 3.3.

Alterado na versão 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** for the rationale).

`socket.`**`sendmsg_afalg`**([*msg*], *, *op*[, *iv*[, *assoclen*[, *flags*]]])
Specialized version of `sendmsg()` for `AF_ALG` socket. Set mode, IV, AEAD associated data length and flags for `AF_ALG` socket.

Disponibilidade: Linux >= 2.6.38.

Novo na versão 3.6.

`socket.`**`sendfile`**(*file*, *offset=0*, *count=None*)
Send a file until EOF is reached by using high-performance `os.sendfile` and return the total number of bytes which were sent. *file* must be a regular file object opened in binary mode. If `os.sendfile` is not available (e.g. Windows) or *file* is not a regular file `send()` will be used instead. *offset* tells from where to start reading the file. If specified, *count* is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is updated on return or also in case of error in which case `file.tell()` can be used to figure out the number of bytes which were sent. The socket must be of `SOCK_STREAM` type. Non-blocking sockets are not supported.

Novo na versão 3.5.

socket.**set_inheritable**(*inheritable*)
Set the *inheritable flag* of the socket's file descriptor or socket's handle.

Novo na versão 3.4.

socket.**setblocking**(*flag*)
Set blocking or non-blocking mode of the socket: if *flag* is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain *settimeout()* calls:

- `sock.setblocking(True)` is equivalent to `sock.settimeout(None)`
- `sock.setblocking(False)` is equivalent to `sock.settimeout(0.0)`

Alterado na versão 3.7: The method no longer applies *SOCK_NONBLOCK* flag on *socket.type*.

socket.**settimeout**(*value*)
Set a timeout on blocking socket operations. The *value* argument can be a nonnegative floating point number expressing seconds, or `None`. If a non-zero value is given, subsequent socket operations will raise a *timeout* exception if the timeout period *value* has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If `None` is given, the socket is put in blocking mode.

For further information, please consult the *notes on socket timeouts*.

Alterado na versão 3.7: The method no longer toggles *SOCK_NONBLOCK* flag on *socket.type*.

socket.**setsockopt**(*level*, *optname*, *value: int*)

socket.**setsockopt**(*level*, *optname*, *value: buffer*)

socket.**setsockopt**(*level*, *optname*, *None*, *optlen: int*)
Set the value of the given socket option (see the Unix manual page *setsockopt(2)*). The needed symbolic constants are defined in the *socket* module (`SO_*` etc.). The value can be an integer, `None` or a *bytes-like object* representing a buffer. In the later case it is up to the caller to ensure that the bytestring contains the proper bits (see the optional built-in module *struct* for a way to encode C structures as bytestrings). When *value* is set to `None`, *optlen* argument is required. It's equivalent to call `setsockopt()` C function with `optval=NULL` and `optlen=optlen`.

Alterado na versão 3.5: Writable *bytes-like object* is now accepted.

Alterado na versão 3.6: setsockopt(level, optname, None, optlen: int) form added.

socket.**shutdown**(*how*)
Shut down one or both halves of the connection. If *how* is `SHUT_RD`, further receives are disallowed. If *how* is `SHUT_WR`, further sends are disallowed. If *how* is `SHUT_RDWR`, further sends and receives are disallowed.

socket.**share**(*process_id*)
Duplicate a socket and prepare it for sharing with a target process. The target process must be provided with *process_id*. The resulting bytes object can then be passed to the target process using some form of interprocess communication and the socket can be recreated there using *fromshare()*. Once this method has been called, it is safe to close the socket since the operating system has already duplicated it for the target process.

Disponibilidade: Windows.

Novo na versão 3.3.

Note that there are no methods `read()` or `write()`; use *recv()* and *send()* without *flags* argument instead.

Socket objects also have these (read-only) attributes that correspond to the values given to the *socket* constructor.

socket.**family**
The socket family.

socket.**type**
The socket type.

socket.**proto**
The socket protocol.

19.2.4 Notas sobre tempo limite de soquete

Um objeto soquete pode estar em um dos três modos: bloqueio, não-bloqueio ou tempo limite. Por padrão, os soquetes sempre são criados no modo de bloqueio, mas isso pode ser alterado chamando *setdefaulttimeout()*.

- In *blocking mode*, operations block until complete or the system returns an error (such as connection timed out).
- In *non-blocking mode*, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately: functions from the *select* can be used to know when and whether a socket is available for reading or writing.
- In *timeout mode*, operations fail if they cannot be completed within the timeout specified for the socket (they raise a *timeout* exception) or if the system returns an error.

Nota: At the operating system level, sockets in *timeout mode* are internally set in non-blocking mode. Also, the blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. This implementation detail can have visible consequences if e.g. you decide to use the *fileno()* of a socket.

Timeouts and the `connect` method

The *connect()* operation is also subject to the timeout setting, and in general it is recommended to call *settimeout()* before calling *connect()* or pass a timeout parameter to *create_connection()*. However, the system network stack may also return a connection timeout error of its own regardless of any Python socket timeout setting.

Timeouts and the `accept` method

If *getdefaulttimeout()* is not *None*, sockets returned by the *accept()* method inherit that timeout. Otherwise, the behaviour depends on settings of the listening socket:

- if the listening socket is in *blocking mode* or in *timeout mode*, the socket returned by *accept()* is in *blocking mode*;
- if the listening socket is in *non-blocking mode*, whether the socket returned by *accept()* is in blocking or non-blocking mode is operating system-dependent. If you want to ensure cross-platform behaviour, it is recommended you manually override this setting.

19.2.5 Exemplo

Here are four minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence *socket()*, *bind()*, *listen()*, *accept()* (possibly repeating the *accept()* to service more than one client), while a client only needs the sequence *socket()*, *connect()*. Also note that the server does not *sendall()*/*recv()* on the socket it is listening on but on the new socket returned by *accept()*.

The first two examples support IPv4 only.

```
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data: break
            conn.sendall(data)
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to the all addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

```
# Echo server program
import socket
import sys

HOST = None               # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
```

(continua na próxima página)

(continuação da página anterior)

```
    try:
        s.bind(sa)
        s.listen(1)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data: break
        conn.send(data)
```

```
# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
with s:
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

The next example shows how to write a very simple network sniffer with raw sockets on Windows. The example requires administrator privileges to modify the interface:

```
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())
```

(continua na próxima página)

(continuação da página anterior)

```
# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a package
print(s.recvfrom(65565))

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

The next example shows how to use the socket interface to communicate to a CAN network using the raw socket protocol. To use CAN with the broadcast manager protocol instead, open a socket with:

```
socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_BCM)
```

After binding (`CAN_RAW`) or connecting (*CAN_BCM*) the socket, you can use the *socket.send()*, and the *socket.recv()* operations (and their counterparts) on the socket object as usual.

This last example might require special privileges:

```
import socket
import struct


# CAN frame packing/unpacking (see 'struct can_frame' in <linux/can.h>)

can_frame_fmt = "=IB3x8s"
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc, data)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
    return (can_id, can_dlc, data[:can_dlc])


# create a raw socket and bind it to the 'vcan0' interface
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW, socket.CAN_RAW)
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)

    print('Received: can_id=%x, can_dlc=%x, data=%s' % dissect_can_frame(cf))

    try:
        s.send(cf)
    except OSError:
```

(continua na próxima página)

(continuação da página anterior)

```
        print('Error sending CAN frame')

    try:
        s.send(build_can_frame(0x01, b'\x01\x02\x03'))
    except OSError:
        print('Error sending CAN frame')
```

Running an example several times with too small delay between executions, could lead to this error:

```
OSError: [Errno 98] Address already in use
```

This is because the previous execution has left the socket in a `TIME_WAIT` state, and can't be immediately reused.

There is a *socket* flag to set, in order to prevent this, `socket.SO_REUSEADDR`:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

the `SO_REUSEADDR` flag tells the kernel to reuse a local socket in `TIME_WAIT` state, without waiting for its natural timeout to expire.

Ver também:

Para uma introdução à programação de socket (em C), veja os seguintes artigos:

- *An Introductory 4.3BSD Interprocess Communication Tutorial*, by Stuart Sechrest
- *An Advanced 4.3BSD Interprocess Communication Tutorial*, by Samuel J. Leffler et al,

both in the UNIX Programmer's Manual, Supplementary Documents 1 (sections PS1:7 and PS1:8). The platform-specific reference material for the various socket-related system calls are also a valuable source of information on the details of socket semantics. For Unix, refer to the manual pages; for Windows, see the WinSock (or Winsock 2) specification. For IPv6-ready APIs, readers may want to refer to **RFC 3493** titled Basic Socket Interface Extensions for IPv6.

19.3 `ssl` — TLS/SSL wrapper for socket objects

Código Fonte: Lib/ssl.py

This module provides access to Transport Layer Security (often known as "Secure Sockets Layer") encryption and peer authentication facilities for network sockets, both client-side and server-side. This module uses the OpenSSL library. It is available on all modern Unix systems, Windows, Mac OS X, and probably additional platforms, as long as OpenSSL is installed on that platform.

Nota: Some behavior may be platform dependent, since calls are made to the operating system socket APIs. The installed version of OpenSSL may also cause variations in behavior. For example, TLSv1.1 and TLSv1.2 come with openssl version 1.0.1.

Aviso: Don't use this module without reading the *Considerações de segurança*. Doing so may lead to a false sense of security, as the default settings of the ssl module are not necessarily appropriate for your application.

This section documents the objects and functions in the `ssl` module; for more general information about TLS, SSL, and certificates, the reader is referred to the documents in the "See Also" section at the bottom.

This module provides a class, *ssl.SSLSocket*, which is derived from the *socket.socket* type, and provides a socket-like wrapper that also encrypts and decrypts the data going over the socket with SSL. It supports additional methods such as `getpeercert()`, which retrieves the certificate of the other side of the connection, and `cipher()`, which retrieves the cipher being used for the secure connection.

For more sophisticated applications, the *ssl.SSLContext* class helps manage settings and certificates, which can then be inherited by SSL sockets created through the *SSLContext.wrap_socket()* method.

Alterado na versão 3.5.3: Updated to support linking with OpenSSL 1.1.0

Alterado na versão 3.6: OpenSSL 0.9.8, 1.0.0 and 1.0.1 are deprecated and no longer supported. In the future the ssl module will require at least OpenSSL 1.0.2 or 1.1.0.

19.3.1 Functions, Constants, and Exceptions

Socket creation

Since Python 3.2 and 2.7.9, it is recommended to use the *SSLContext.wrap_socket()* of an *SSLContext* instance to wrap sockets as *SSLSocket* objects. The helper functions *create_default_context()* returns a new context with secure default settings. The old *wrap_socket()* function is deprecated since it is both inefficient and has no support for server name indication (SNI) and hostname matching.

Client socket example with default context and IPv4/IPv6 dual stack:

```
import socket
import ssl

hostname = 'www.python.org'
context = ssl.create_default_context()

with socket.create_connection((hostname, 443)) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

Client socket example with custom context and IPv4:

```
hostname = 'www.python.org'
# PROTOCOL_TLS_CLIENT requires valid cert chain and hostname
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.load_verify_locations('path/to/cabundle.pem')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

Server socket example listening on localhost IPv4:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain('/path/to/certchain.pem', '/path/to/private.key')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    sock.bind(('127.0.0.1', 8443))
    sock.listen(5)
    with context.wrap_socket(sock, server_side=True) as ssock:
```

(continua na próxima página)

(continuação da página anterior)

```
        conn, addr = ssock.accept()
        ...
```

Context creation

A convenience function helps create *SSLContext* objects for common purposes.

ssl.**create_default_context**(*purpose=Purpose.SERVER_AUTH*, *cafile=None*, *capath=None*, *cadata=None*)

Return a new *SSLContext* object with default settings for the given *purpose*. The settings are chosen by the *ssl* module, and usually represent a higher security level than when calling the *SSLContext* constructor directly.

cafile, *capath*, *cadata* represent optional CA certificates to trust for certificate verification, as in *SSLContext.load_verify_locations()*. If all three are *None*, this function can choose to trust the system's default CA certificates instead.

The settings are: *PROTOCOL_TLS*, *OP_NO_SSLv2*, and *OP_NO_SSLv3* with high encryption cipher suites without RC4 and without unauthenticated cipher suites. Passing *SERVER_AUTH* as *purpose* sets *verify_mode* to *CERT_REQUIRED* and either loads CA certificates (when at least one of *cafile*, *capath* or *cadata* is given) or uses *SSLContext.load_default_certs()* to load default CA certificates.

Nota: The protocol, options, cipher and other settings may change to more restrictive values anytime without prior deprecation. The values represent a fair balance between compatibility and security.

If your application needs specific settings, you should create a *SSLContext* and apply the settings yourself.

Nota: If you find that when certain older clients or servers attempt to connect with a *SSLContext* created by this function that they get an error stating "Protocol or cipher suite mismatch", it may be that they only support SSL3.0 which this function excludes using the *OP_NO_SSLv3*. SSL3.0 is widely considered to be completely broken. If you still wish to continue to use this function but still allow SSL 3.0 connections you can re-enable them using:

```
ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
ctx.options &= ~ssl.OP_NO_SSLv3
```

Novo na versão 3.4.

Alterado na versão 3.4.4: RC4 was dropped from the default cipher string.

Alterado na versão 3.6: ChaCha20/Poly1305 was added to the default cipher string.

3DES was dropped from the default cipher string.

Exceções

exception ssl.**SSLError**
Raised to signal an error from the underlying SSL implementation (currently provided by the OpenSSL library). This signifies some problem in the higher-level encryption and authentication layer that's superimposed on the underlying network connection. This error is a subtype of *OSError*. The error code and message of *SSLError* instances are provided by the OpenSSL library.

Alterado na versão 3.3: *SSLError* used to be a subtype of *socket.error*.

library
A string mnemonic designating the OpenSSL submodule in which the error occurred, such as SSL, PEM or X509. The range of possible values depends on the OpenSSL version.

Novo na versão 3.3.

reason
A string mnemonic designating the reason this error occurred, for example CERTIFICATE_VERIFY_FAILED. The range of possible values depends on the OpenSSL version.

Novo na versão 3.3.

exception ssl.**SSLZeroReturnError**
A subclass of *SSLError* raised when trying to read or write and the SSL connection has been closed cleanly. Note that this doesn't mean that the underlying transport (read TCP) has been closed.

Novo na versão 3.3.

exception ssl.**SSLWantReadError**
A subclass of *SSLError* raised by a *non-blocking SSL socket* when trying to read or write data, but more data needs to be received on the underlying TCP transport before the request can be fulfilled.

Novo na versão 3.3.

exception ssl.**SSLWantWriteError**
A subclass of *SSLError* raised by a *non-blocking SSL socket* when trying to read or write data, but more data needs to be sent on the underlying TCP transport before the request can be fulfilled.

Novo na versão 3.3.

exception ssl.**SSLSyscallError**
A subclass of *SSLError* raised when a system error was encountered while trying to fulfill an operation on a SSL socket. Unfortunately, there is no easy way to inspect the original errno number.

Novo na versão 3.3.

exception ssl.**SSLEOFError**
A subclass of *SSLError* raised when the SSL connection has been terminated abruptly. Generally, you shouldn't try to reuse the underlying transport when this error is encountered.

Novo na versão 3.3.

exception ssl.**SSLCertVerificationError**
A subclass of *SSLError* raised when certificate validation has failed.

Novo na versão 3.7.

verify_code
A numeric error number that denotes the verification error.

verify_message
A human readable string of the verification error.

exception ssl.**CertificateError**
An alias for *SSLCertVerificationError*.

Alterado na versão 3.7: The exception is now an alias for *SSLCertVerificationError*.

Random generation

ssl.**RAND_bytes**(*num*)
Return *num* cryptographically strong pseudo-random bytes. Raises an *SSLError* if the PRNG has not been seeded with enough data or if the operation is not supported by the current RAND method. *RAND_status()* can be used to check the status of the PRNG and *RAND_add()* can be used to seed the PRNG.

For almost all applications *os.urandom()* is preferable.

Read the Wikipedia article, Cryptographically secure pseudorandom number generator (CSPRNG), to get the requirements of a cryptographically generator.

Novo na versão 3.3.

ssl.**RAND_pseudo_bytes**(*num*)
Return (bytes, is_cryptographic): bytes are *num* pseudo-random bytes, is_cryptographic is `True` if the bytes generated are cryptographically strong. Raises an *SSLError* if the operation is not supported by the current RAND method.

Generated pseudo-random byte sequences will be unique if they are of sufficient length, but are not necessarily unpredictable. They can be used for non-cryptographic purposes and for certain purposes in cryptographic protocols, but usually not for key generation etc.

For almost all applications *os.urandom()* is preferable.

Novo na versão 3.3.

Obsoleto desde a versão 3.6: OpenSSL has deprecated *ssl.RAND_pseudo_bytes()*, use *ssl.RAND_bytes()* instead.

ssl.**RAND_status**()
Return `True` if the SSL pseudo-random number generator has been seeded with 'enough' randomness, and `False` otherwise. You can use *ssl.RAND_egd()* and *ssl.RAND_add()* to increase the randomness of the pseudo-random number generator.

ssl.**RAND_egd**(*path*)
If you are running an entropy-gathering daemon (EGD) somewhere, and *path* is the pathname of a socket connection open to it, this will read 256 bytes of randomness from the socket, and add it to the SSL pseudo-random number generator to increase the security of generated secret keys. This is typically only necessary on systems without better sources of randomness.

See http://egd.sourceforge.net/ or http://prngd.sourceforge.net/ for sources of entropy-gathering daemons.

Availability: not available with LibreSSL and OpenSSL > 1.1.0.

ssl.**RAND_add**(*bytes*, *entropy*)
Mix the given *bytes* into the SSL pseudo-random number generator. The parameter *entropy* (a float) is a lower bound on the entropy contained in string (so you can always use `0.0`). See **RFC 1750** for more information on sources of entropy.

Alterado na versão 3.5: Writable *bytes-like object* is now accepted.

Manipulação de certificados

ssl.**match_hostname**(*cert*, *hostname*)
Verify that *cert* (in decoded format as returned by *SSLSocket.getpeercert()*) matches the given *hostname*. The rules applied are those for checking the identity of HTTPS servers as outlined in **RFC 2818**, **RFC 5280** and **RFC 6125**. In addition to HTTPS, this function should be suitable for checking the identity of servers in various SSL-based protocols such as FTPS, IMAPS, POPS and others.

CertificateError is raised on failure. On success, the function returns nothing:

```
>>> cert = {'subject': ((('commonName', 'example.com'),),)}
>>> ssl.match_hostname(cert, "example.com")
>>> ssl.match_hostname(cert, "example.org")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/py3k/Lib/ssl.py", line 130, in match_hostname
ssl.CertificateError: hostname 'example.org' doesn't match 'example.com'
```

Novo na versão 3.2.

Alterado na versão 3.3.3: The function now follows **RFC 6125**, section 6.4.3 and does neither match multiple wildcards (e.g. `*.*.com` or `*a*.example.org`) nor a wildcard inside an internationalized domain names (IDN) fragment. IDN A-labels such as `www*.xn--pthon-kva.org` are still supported, but `x*.python.org` no longer matches `xn--tda.python.org`.

Alterado na versão 3.5: Matching of IP addresses, when present in the subjectAltName field of the certificate, is now supported.

Alterado na versão 3.7: The function is no longer used to TLS connections. Hostname matching is now performed by OpenSSL.

Allow wildcard when it is the leftmost and the only character in that segment. Partial wildcards like `www*.example.com` are no longer supported.

Obsoleto desde a versão 3.7.

ssl.**cert_time_to_seconds**(*cert_time*)
Return the time in seconds since the Epoch, given the `cert_time` string representing the "notBefore" or "notAfter" date from a certificate in `"%b %d %H:%M:%S %Y %Z"` strptime format (C locale).

Here's an example:

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

"notBefore" or "notAfter" dates must use GMT (**RFC 5280**).

Alterado na versão 3.5: Interpret the input time as a time in UTC as specified by 'GMT' timezone in the input string. Local timezone was used previously. Return an integer (no fractions of a second in the input format)

ssl.**get_server_certificate**(*addr*, *ssl_version=PROTOCOL_TLS*, *ca_certs=None*)
Given the address `addr` of an SSL-protected server, as a (*hostname*, *port-number*) pair, fetches the server's certificate, and returns it as a PEM-encoded string. If `ssl_version` is specified, uses that version of the SSL protocol to attempt to connect to the server. If `ca_certs` is specified, it should be a file containing a list of root certificates,

the same format as used for the same parameter in *SSLContext.wrap_socket()*. The call will attempt to validate the server certificate against that set of root certificates, and will fail if the validation attempt fails.

Alterado na versão 3.3: This function is now IPv6-compatible.

Alterado na versão 3.5: The default *ssl_version* is changed from *PROTOCOL_SSLv3* to *PROTOCOL_TLS* for maximum compatibility with modern servers.

ssl.**DER_cert_to_PEM_cert**(*DER_cert_bytes*)
Given a certificate as a DER-encoded blob of bytes, returns a PEM-encoded string version of the same certificate.

ssl.**PEM_cert_to_DER_cert**(*PEM_cert_string*)
Given a certificate as an ASCII PEM string, returns a DER-encoded sequence of bytes for that same certificate.

ssl.**get_default_verify_paths**()
Returns a named tuple with paths to OpenSSL's default cafile and capath. The paths are the same as used by *SSLContext.set_default_verify_paths()*. The return value is a *named tuple* `DefaultVerifyPaths`:

- `cafile` - resolved path to cafile or `None` if the file doesn't exist,
- `capath` - resolved path to capath or `None` if the directory doesn't exist,
- `openssl_cafile_env` - OpenSSL's environment key that points to a cafile,
- `openssl_cafile` - hard coded path to a cafile,
- `openssl_capath_env` - OpenSSL's environment key that points to a capath,
- `openssl_capath` - hard coded path to a capath directory

Availability: LibreSSL ignores the environment vars `openssl_cafile_env` and `openssl_capath_env`.

Novo na versão 3.4.

ssl.**enum_certificates**(*store_name*)
Retrieve certificates from Windows' system cert store. *store_name* may be one of `CA`, `ROOT` or `MY`. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either `x509_asn` for X.509 ASN.1 data or `pkcs_7_asn` for PKCS#7 ASN.1 data. Trust specifies the purpose of the certificate as a set of OIDS or exactly `True` if the certificate is trustworthy for all purposes.

Exemplo:

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

Disponibilidade: Windows.

Novo na versão 3.4.

ssl.**enum_crls**(*store_name*)
Retrieve CRLs from Windows' system cert store. *store_name* may be one of `CA`, `ROOT` or `MY`. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either `x509_asn` for X.509 ASN.1 data or `pkcs_7_asn` for PKCS#7 ASN.1 data.

Disponibilidade: Windows.

Novo na versão 3.4.

ssl.**wrap_socket**(*sock*, *keyfile=None*, *certfile=None*, *server_side=False*, *cert_reqs=CERT_NONE*, *ssl_version=PROTOCOL_TLS*, *ca_certs=None*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *ciphers=None*)

Takes an instance `sock` of *socket.socket*, and returns an instance of *ssl.SSLSocket*, a subtype of *socket.socket*, which wraps the underlying socket in an SSL context. `sock` must be a *SOCK_STREAM* socket; other socket types are unsupported.

Internally, function creates a *SSLContext* with protocol *ssl_version* and *SSLContext.options* set to *cert_reqs*. If parameters *keyfile*, *certfile*, *ca_certs* or *ciphers* are set, then the values are passed to *SSLContext.load_cert_chain()*, *SSLContext.load_verify_locations()*, and *SSLContext.set_ciphers()*.

The arguments *server_side*, *do_handshake_on_connect*, and *suppress_ragged_eofs* have the same meaning as *SSLContext.wrap_socket()*.

Obsoleto desde a versão 3.7: Since Python 3.2 and 2.7.9, it is recommended to use the *SSLContext.wrap_socket()* instead of *wrap_socket()*. The top-level function is limited and creates an insecure client socket without server name indication or hostname matching.

Constantes

All constants are now *enum.IntEnum* or *enum.IntFlag* collections.

Novo na versão 3.6.

ssl.**CERT_NONE**

Possible value for *SSLContext.verify_mode*, or the `cert_reqs` parameter to *wrap_socket()*. Except for *PROTOCOL_TLS_CLIENT*, it is the default mode. With client-side sockets, just about any cert is accepted. Validation errors, such as untrusted or expired cert, are ignored and do not abort the TLS/SSL handshake.

In server mode, no certificate is requested from the client, so the client does not send any for client cert authentication.

See the discussion of *Considerações de segurança* below.

ssl.**CERT_OPTIONAL**

Possible value for *SSLContext.verify_mode*, or the `cert_reqs` parameter to *wrap_socket()*. In client mode, *CERT_OPTIONAL* has the same meaning as *CERT_REQUIRED*. It is recommended to use *CERT_REQUIRED* for client-side sockets instead.

In server mode, a client certificate request is sent to the client. The client may either ignore the request or send a certificate in order perform TLS client cert authentication. If the client chooses to send a certificate, it is verified. Any verification error immediately aborts the TLS handshake.

Use of this setting requires a valid set of CA certificates to be passed, either to *SSLContext.load_verify_locations()* or as a value of the `ca_certs` parameter to *wrap_socket()*.

ssl.**CERT_REQUIRED**

Possible value for *SSLContext.verify_mode*, or the `cert_reqs` parameter to *wrap_socket()*. In this mode, certificates are required from the other side of the socket connection; an *SSLError* will be raised if no certificate is provided, or if its validation fails. This mode is **not** sufficient to verify a certificate in client mode as it does not match hostnames. *check_hostname* must be enabled as well to verify the authenticity of a cert. *PROTOCOL_TLS_CLIENT* uses *CERT_REQUIRED* and enables *check_hostname* by default.

With server socket, this mode provides mandatory TLS client cert authentication. A client certificate request is sent to the client and the client must provide a valid and trusted certificate.

Use of this setting requires a valid set of CA certificates to be passed, either to *SSLContext.load_verify_locations()* or as a value of the `ca_certs` parameter to *wrap_socket()*.

class ssl.**VerifyMode**
enum.IntEnum collection of CERT_* constants.

Novo na versão 3.6.

ssl.**VERIFY_DEFAULT**
Possible value for *SSLContext.verify_flags*. In this mode, certificate revocation lists (CRLs) are not checked. By default OpenSSL does neither require nor verify CRLs.

Novo na versão 3.4.

ssl.**VERIFY_CRL_CHECK_LEAF**
Possible value for *SSLContext.verify_flags*. In this mode, only the peer cert is checked but none of the intermediate CA certificates. The mode requires a valid CRL that is signed by the peer cert's issuer (its direct ancestor CA). If no proper CRL has has been loaded with *SSLContext.load_verify_locations*, validation will fail.

Novo na versão 3.4.

ssl.**VERIFY_CRL_CHECK_CHAIN**
Possible value for *SSLContext.verify_flags*. In this mode, CRLs of all certificates in the peer cert chain are checked.

Novo na versão 3.4.

ssl.**VERIFY_X509_STRICT**
Possible value for *SSLContext.verify_flags* to disable workarounds for broken X.509 certificates.

Novo na versão 3.4.

ssl.**VERIFY_X509_TRUSTED_FIRST**
Possible value for *SSLContext.verify_flags*. It instructs OpenSSL to prefer trusted certificates when building the trust chain to validate a certificate. This flag is enabled by default.

Novo na versão 3.4.4.

class ssl.**VerifyFlags**
enum.IntFlag collection of VERIFY_* constants.

Novo na versão 3.6.

ssl.**PROTOCOL_TLS**
Selects the highest protocol version that both the client and server support. Despite the name, this option can select both "SSL" and "TLS" protocols.

Novo na versão 3.6.

ssl.**PROTOCOL_TLS_CLIENT**
Auto-negotiate the highest protocol version like *PROTOCOL_TLS*, but only support client-side *SSLSocket* connections. The protocol enables *CERT_REQUIRED* and *check_hostname* by default.

Novo na versão 3.6.

ssl.**PROTOCOL_TLS_SERVER**
Auto-negotiate the highest protocol version like *PROTOCOL_TLS*, but only support server-side *SSLSocket* connections.

Novo na versão 3.6.

ssl.**PROTOCOL_SSLv23**
Alias for *PROTOCOL_TLS*.

Obsoleto desde a versão 3.6: Use *PROTOCOL_TLS* instead.

ssl.**PROTOCOL_SSLv2**
Selects SSL version 2 as the channel encryption protocol.

This protocol is not available if OpenSSL is compiled with the `OPENSSL_NO_SSL2` flag.

> **Aviso:** SSL version 2 is insecure. Its use is highly discouraged.

Obsoleto desde a versão 3.6: OpenSSL has removed support for SSLv2.

ssl.**PROTOCOL_SSLv3**
Selects SSL version 3 as the channel encryption protocol.

This protocol is not be available if OpenSSL is compiled with the `OPENSSL_NO_SSLv3` flag.

> **Aviso:** SSL version 3 is insecure. Its use is highly discouraged.

Obsoleto desde a versão 3.6: OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS* with flags like *OP_NO_SSLv3* instead.

ssl.**PROTOCOL_TLSv1**
Selects TLS version 1.0 as the channel encryption protocol.

Obsoleto desde a versão 3.6: OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS* with flags like *OP_NO_SSLv3* instead.

ssl.**PROTOCOL_TLSv1_1**
Selects TLS version 1.1 as the channel encryption protocol. Available only with openssl version 1.0.1+.

Novo na versão 3.4.

Obsoleto desde a versão 3.6: OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS* with flags like *OP_NO_SSLv3* instead.

ssl.**PROTOCOL_TLSv1_2**
Selects TLS version 1.2 as the channel encryption protocol. This is the most modern version, and probably the best choice for maximum protection, if both sides can speak it. Available only with openssl version 1.0.1+.

Novo na versão 3.4.

Obsoleto desde a versão 3.6: OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS* with flags like *OP_NO_SSLv3* instead.

ssl.**OP_ALL**
Enables workarounds for various bugs present in other SSL implementations. This option is set by default. It does not necessarily set the same flags as OpenSSL's `SSL_OP_ALL` constant.

Novo na versão 3.2.

ssl.**OP_NO_SSLv2**
Prevents an SSLv2 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing SSLv2 as the protocol version.

Novo na versão 3.2.

Obsoleto desde a versão 3.6: SSLv2 is deprecated

ssl.**OP_NO_SSLv3**
Prevents an SSLv3 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing SSLv3 as the protocol version.

Novo na versão 3.2.

Obsoleto desde a versão 3.6: SSLv3 is deprecated

ssl.**OP_NO_TLSv1**
Prevents a TLSv1 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1 as the protocol version.

Novo na versão 3.2.

Obsoleto desde a versão 3.7: The option is deprecated since OpenSSL 1.1.0, use the new *SSLContext.minimum_version* and *SSLContext.maximum_version* instead.

ssl.**OP_NO_TLSv1_1**
Prevents a TLSv1.1 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1.1 as the protocol version. Available only with openssl version 1.0.1+.

Novo na versão 3.4.

Obsoleto desde a versão 3.7: The option is deprecated since OpenSSL 1.1.0.

ssl.**OP_NO_TLSv1_2**
Prevents a TLSv1.2 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1.2 as the protocol version. Available only with openssl version 1.0.1+.

Novo na versão 3.4.

Obsoleto desde a versão 3.7: The option is deprecated since OpenSSL 1.1.0.

ssl.**OP_NO_TLSv1_3**
Prevents a TLSv1.3 connection. This option is only applicable in conjunction with *PROTOCOL_TLS*. It prevents the peers from choosing TLSv1.3 as the protocol version. TLS 1.3 is available with OpenSSL 1.1.1 or later. When Python has been compiled against an older version of OpenSSL, the flag defaults to *0*.

Novo na versão 3.7.

Obsoleto desde a versão 3.7: The option is deprecated since OpenSSL 1.1.0. It was added to 2.7.15, 3.6.3 and 3.7.0 for backwards compatibility with OpenSSL 1.0.2.

ssl.**OP_NO_RENEGOTIATION**
Disable all renegotiation in TLSv1.2 and earlier. Do not send HelloRequest messages, and ignore renegotiation requests via ClientHello.

This option is only available with OpenSSL 1.1.0h and later.

Novo na versão 3.7.

ssl.**OP_CIPHER_SERVER_PREFERENCE**
Use the server's cipher ordering preference, rather than the client's. This option has no effect on client sockets and SSLv2 server sockets.

Novo na versão 3.3.

ssl.**OP_SINGLE_DH_USE**
Prevents re-use of the same DH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

Novo na versão 3.3.

ssl.**OP_SINGLE_ECDH_USE**
Prevents re-use of the same ECDH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

Novo na versão 3.3.

ssl.**OP_ENABLE_MIDDLEBOX_COMPAT**
Send dummy Change Cipher Spec (CCS) messages in TLS 1.3 handshake to make a TLS 1.3 connection look more like a TLS 1.2 connection.

This option is only available with OpenSSL 1.1.1 and later.

Novo na versão 3.8.

ssl.**OP_NO_COMPRESSION**
Disable compression on the SSL channel. This is useful if the application protocol supports its own compression scheme.

This option is only available with OpenSSL 1.0.0 and later.

Novo na versão 3.3.

class ssl.**Options**
enum.IntFlag collection of OP_* constants.

ssl.**OP_NO_TICKET**
Prevent client side from requesting a session ticket.

Novo na versão 3.6.

ssl.**HAS_ALPN**
Whether the OpenSSL library has built-in support for the *Application-Layer Protocol Negotiation* TLS extension as described in **RFC 7301**.

Novo na versão 3.5.

ssl.**HAS_NEVER_CHECK_COMMON_NAME**
Whether the OpenSSL library has built-in support not checking subject common name and *SSLContext.hostname_checks_common_name* is writeable.

Novo na versão 3.7.

ssl.**HAS_ECDH**
Whether the OpenSSL library has built-in support for the Elliptic Curve-based Diffie-Hellman key exchange. This should be true unless the feature was explicitly disabled by the distributor.

Novo na versão 3.3.

ssl.**HAS_SNI**
Whether the OpenSSL library has built-in support for the *Server Name Indication* extension (as defined in **RFC 6066**).

Novo na versão 3.2.

ssl.**HAS_NPN**
Whether the OpenSSL library has built-in support for the *Next Protocol Negotiation* as described in the Application Layer Protocol Negotiation. When true, you can use the *SSLContext.set_npn_protocols()* method to advertise which protocols you want to support.

Novo na versão 3.3.

ssl.**HAS_SSLv2**
Whether the OpenSSL library has built-in support for the SSL 2.0 protocol.

Novo na versão 3.7.

ssl.**HAS_SSLv3**
Whether the OpenSSL library has built-in support for the SSL 3.0 protocol.

Novo na versão 3.7.

ssl.**HAS_TLSv1**
Whether the OpenSSL library has built-in support for the TLS 1.0 protocol.

Novo na versão 3.7.

ssl.**HAS_TLSv1_1**
Whether the OpenSSL library has built-in support for the TLS 1.1 protocol.

Novo na versão 3.7.

ssl.**HAS_TLSv1_2**
Whether the OpenSSL library has built-in support for the TLS 1.2 protocol.

Novo na versão 3.7.

ssl.**HAS_TLSv1_3**
Whether the OpenSSL library has built-in support for the TLS 1.3 protocol.

Novo na versão 3.7.

ssl.**CHANNEL_BINDING_TYPES**
List of supported TLS channel binding types. Strings in this list can be used as arguments to *SSLSocket.get_channel_binding()*.

Novo na versão 3.3.

ssl.**OPENSSL_VERSION**
The version string of the OpenSSL library loaded by the interpreter:

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

Novo na versão 3.2.

ssl.**OPENSSL_VERSION_INFO**
A tuple of five integers representing version information about the OpenSSL library:

```
>>> ssl.OPENSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

Novo na versão 3.2.

ssl.**OPENSSL_VERSION_NUMBER**
The raw version number of the OpenSSL library, as a single integer:

```
>>> ssl.OPENSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x100020bf'
```

Novo na versão 3.2.

ssl.**ALERT_DESCRIPTION_HANDSHAKE_FAILURE**
ssl.**ALERT_DESCRIPTION_INTERNAL_ERROR**
ALERT_DESCRIPTION_*
Alert Descriptions from **RFC 5246** and others. The IANA TLS Alert Registry contains this list and references to the RFCs where their meaning is defined.

Used as the return value of the callback function in *SSLContext.set_servername_callback()*.

Novo na versão 3.4.

class ssl.**AlertDescription**
enum.IntEnum collection of ALERT_DESCRIPTION_* constants.

Novo na versão 3.6.

Purpose.**SERVER_AUTH**
Option for *create_default_context()* and *SSLContext.load_default_certs()*. This value indicates that the context may be used to authenticate Web servers (therefore, it will be used to create client-side sockets).

Novo na versão 3.4.

Purpose.**CLIENT_AUTH**
Option for *create_default_context()* and *SSLContext.load_default_certs()*. This value indicates that the context may be used to authenticate Web clients (therefore, it will be used to create server-side sockets).

Novo na versão 3.4.

class ssl.**SSLErrorNumber**
enum.IntEnum collection of SSL_ERROR_* constants.

Novo na versão 3.6.

class ssl.**TLSVersion**
enum.IntEnum collection of SSL and TLS versions for *SSLContext.maximum_version* and *SSLContext.minimum_version*.

Novo na versão 3.7.

TLSVersion.**MINIMUM_SUPPORTED**

TLSVersion.**MAXIMUM_SUPPORTED**
The minimum or maximum supported SSL or TLS version. These are magic constants. Their values don't reflect the lowest and highest available TLS/SSL versions.

TLSVersion.**SSLv3**

TLSVersion.**TLSv1**

TLSVersion.**TLSv1_1**

TLSVersion.**TLSv1_2**

TLSVersion.**TLSv1_3**
SSL 3.0 to TLS 1.3.

19.3.2 SSL Sockets

class ssl.**SSLSocket**(*socket.socket*)
SSL sockets provide the following methods of *Socket Objects*:

- *accept()*
- *bind()*
- *close()*
- *connect()*
- *detach()*
- *fileno()*

- *getpeername()*, *getsockname()*
- *getsockopt()*, *setsockopt()*
- *gettimeout()*, *settimeout()*, *setblocking()*
- *listen()*
- *makefile()*
- *recv()*, *recv_into()* (but passing a non-zero `flags` argument is not allowed)
- *send()*, *sendall()* (with the same limitation)
- *sendfile()* (but *os.sendfile* will be used for plain-text sockets only, else *send()* will be used)
- *shutdown()*

However, since the SSL (and TLS) protocol has its own framing atop of TCP, the SSL sockets abstraction can, in certain respects, diverge from the specification of normal, OS-level sockets. See especially the *notes on non-blocking sockets*.

Instances of *SSLSocket* must be created using the *SSLContext.wrap_socket()* method.

Alterado na versão 3.5: The `sendfile()` method was added.

Alterado na versão 3.5: The `shutdown()` does not reset the socket timeout each time bytes are received or sent. The socket timeout is now to maximum total duration of the shutdown.

Obsoleto desde a versão 3.6: It is deprecated to create a *SSLSocket* instance directly, use *SSLContext.wrap_socket()* to wrap a socket.

Alterado na versão 3.7: *SSLSocket* instances must to created with *wrap_socket()*. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

SSL sockets also have the following additional methods and attributes:

`SSLSocket.`**`read`**(*len=1024*, *buffer=None*)

Read up to *len* bytes of data from the SSL socket and return the result as a `bytes` instance. If *buffer* is specified, then read into the buffer instead, and return the number of bytes read.

Raise *SSLWantReadError* or *SSLWantWriteError* if the socket is *non-blocking* and the read would block.

As at any time a re-negotiation is possible, a call to *read()* can also cause write operations.

Alterado na versão 3.5: The socket timeout is no more reset each time bytes are received or sent. The socket timeout is now to maximum total duration to read up to *len* bytes.

Obsoleto desde a versão 3.6: Use `recv()` instead of *read()*.

`SSLSocket.`**`write`**(*buf*)

Write *buf* to the SSL socket and return the number of bytes written. The *buf* argument must be an object supporting the buffer interface.

Raise *SSLWantReadError* or *SSLWantWriteError* if the socket is *non-blocking* and the write would block.

As at any time a re-negotiation is possible, a call to *write()* can also cause read operations.

Alterado na versão 3.5: The socket timeout is no more reset each time bytes are received or sent. The socket timeout is now to maximum total duration to write *buf*.

Obsoleto desde a versão 3.6: Use `send()` instead of *write()*.

Nota: The *read()* and *write()* methods are the low-level methods that read and write unencrypted, application-level data and decrypt/encrypt it to encrypted, wire-level data. These methods require an active SSL connection, i.e. the handshake was completed and *SSLSocket.unwrap()* was not called.

Normally you should use the socket API methods like *recv()* and *send()* instead of these methods.

SSLSocket.**do_handshake**()
Perform the SSL setup handshake.

Alterado na versão 3.4: The handshake method also performs *match_hostname()* when the *check_hostname* attribute of the socket's *context* is true.

Alterado na versão 3.5: The socket timeout is no more reset each time bytes are received or sent. The socket timeout is now to maximum total duration of the handshake.

Alterado na versão 3.7: Hostname or IP address is matched by OpenSSL during handshake. The function *match_hostname()* is no longer used. In case OpenSSL refuses a hostname or IP address, the handshake is aborted early and a TLS alert message is send to the peer.

SSLSocket.**getpeercert**(*binary_form=False*)
If there is no certificate for the peer on the other end of the connection, return None. If the SSL handshake hasn't been done yet, raise *ValueError*.

If the binary_form parameter is *False*, and a certificate was received from the peer, this method returns a *dict* instance. If the certificate was not validated, the dict is empty. If the certificate was validated, it returns a dict with several keys, amongst them subject (the principal for which the certificate was issued) and issuer (the principal issuing the certificate). If a certificate contains an instance of the *Subject Alternative Name* extension (see **RFC 3280**), there will also be a subjectAltName key in the dictionary.

The subject and issuer fields are tuples containing the sequence of relative distinguished names (RDNs) given in the certificate's data structure for the respective fields, and each RDN is a sequence of name-value pairs. Here is a real-world example:

```
{'issuer': ((('countryName', 'IL'),),
            (('organizationName', 'StartCom Ltd.'),),
            (('organizationalUnitName',
              'Secure Digital Certificate Signing'),),
            (('commonName',
              'StartCom Class 2 Primary Intermediate Server CA'),)),
 'notAfter': 'Nov 22 08:15:19 2013 GMT',
 'notBefore': 'Nov 21 03:09:52 2011 GMT',
 'serialNumber': '95F0',
 'subject': ((('description', '571208-SLe257oHY9fVQ07Z'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
             (('organizationName', 'Electronic Frontier Foundation, Inc.'),),
             (('commonName', '*.eff.org'),),
             (('emailAddress', 'hostmaster@eff.org'),)),
 'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
 'version': 3}
```

Nota: To validate a certificate for a particular service, you can use the *match_hostname()* function.

If the binary_form parameter is *True*, and a certificate was provided, this method returns the DER-encoded form of the entire certificate as a sequence of bytes, or *None* if the peer did not provide a certificate. Whether the peer provides a certificate depends on the SSL socket's role:

- for a client SSL socket, the server will always provide a certificate, regardless of whether validation was required;

- for a server SSL socket, the client will only provide a certificate when requested by the server; therefore *getpeercert()* will return *None* if you used *CERT_NONE* (rather than *CERT_OPTIONAL* or *CERT_REQUIRED*).

Alterado na versão 3.2: The returned dictionary includes additional items such as `issuer` and `notBefore`.

Alterado na versão 3.4: *ValueError* is raised when the handshake isn't done. The returned dictionary includes additional X509v3 extension items such as `crlDistributionPoints`, `caIssuers` and `OCSP` URIs.

Alterado na versão 3.7.6: IPv6 address strings no longer have a trailing new line.

`SSLSocket.`**`cipher`**`()`
Returns a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used. If no connection has been established, returns `None`.

`SSLSocket.`**`shared_ciphers`**`()`
Return the list of ciphers shared by the client during the handshake. Each entry of the returned list is a three-value tuple containing the name of the cipher, the version of the SSL protocol that defines its use, and the number of secret bits the cipher uses. *shared_ciphers()* returns `None` if no connection has been established or the socket is a client socket.

Novo na versão 3.5.

`SSLSocket.`**`compression`**`()`
Return the compression algorithm being used as a string, or `None` if the connection isn't compressed.

If the higher-level protocol supports its own compression mechanism, you can use *OP_NO_COMPRESSION* to disable SSL-level compression.

Novo na versão 3.3.

`SSLSocket.`**`get_channel_binding`**`(`*cb_type="tls-unique"*`)`
Get channel binding data for current connection, as a bytes object. Returns `None` if not connected or the handshake has not been completed.

The *cb_type* parameter allow selection of the desired channel binding type. Valid channel binding types are listed in the *CHANNEL_BINDING_TYPES* list. Currently only the 'tls-unique' channel binding, defined by **RFC 5929**, is supported. *ValueError* will be raised if an unsupported channel binding type is requested.

Novo na versão 3.3.

`SSLSocket.`**`selected_alpn_protocol`**`()`
Return the protocol that was selected during the TLS handshake. If *SSLContext.set_alpn_protocols()* was not called, if the other party does not support ALPN, if this socket does not support any of the client's proposed protocols, or if the handshake has not happened yet, `None` is returned.

Novo na versão 3.5.

`SSLSocket.`**`selected_npn_protocol`**`()`
Return the higher-level protocol that was selected during the TLS/SSL handshake. If *SSLContext.set_npn_protocols()* was not called, or if the other party does not support NPN, or if the handshake has not yet happened, this will return `None`.

Novo na versão 3.3.

`SSLSocket.`**`unwrap`**`()`
Performs the SSL shutdown handshake, which removes the TLS layer from the underlying socket, and returns the underlying socket object. This can be used to go from encrypted operation over a connection to unencrypted. The returned socket should always be used for further communication with the other side of the connection, rather than the original socket.

`SSLSocket.`**`verify_client_post_handshake`**`()`
Requests post-handshake authentication (PHA) from a TLS 1.3 client. PHA can only be initiated for a TLS 1.3

connection from a server-side socket, after the initial TLS handshake and with PHA enabled on both sides, see *SSLContext.post_handshake_auth*.

The method does not perform a cert exchange immediately. The server-side sends a CertificateRequest during the next write event and expects the client to respond with a certificate on the next read event.

If any precondition isn't met (e.g. not TLS 1.3, PHA not enabled), an *SSLError* is raised.

Nota: Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the method raises *NotImplementedError*.

Novo na versão 3.7.1.

SSLSocket.**version**()
Return the actual SSL protocol version negotiated by the connection as a string, or `None` is no secure connection is established. As of this writing, possible return values include `"SSLv2"`, `"SSLv3"`, `"TLSv1"`, `"TLSv1.1"` and `"TLSv1.2"`. Recent OpenSSL versions may define more return values.

Novo na versão 3.5.

SSLSocket.**pending**()
Returns the number of already decrypted bytes available for read, pending on the connection.

SSLSocket.**context**
The *SSLContext* object this SSL socket is tied to. If the SSL socket was created using the deprecated *wrap_socket()* function (rather than *SSLContext.wrap_socket()*), this is a custom context object created for this SSL socket.

Novo na versão 3.2.

SSLSocket.**server_side**
A boolean which is `True` for server-side sockets and `False` for client-side sockets.

Novo na versão 3.2.

SSLSocket.**server_hostname**
Hostname of the server: *str* type, or `None` for server-side socket or if the hostname was not specified in the constructor.

Novo na versão 3.2.

Alterado na versão 3.7: The attribute is now always ASCII text. When `server_hostname` is an internationalized domain name (IDN), this attribute now stores the A-label form (`"xn--pythn-mua.org"`), rather than the U-label form (`"pythön.org"`).

SSLSocket.**session**
The *SSLSession* for this SSL connection. The session is available for client and server side sockets after the TLS handshake has been performed. For client sockets the session can be set before *do_handshake()* has been called to reuse a session.

Novo na versão 3.6.

SSLSocket.**session_reused**
Novo na versão 3.6.

19.3.3 SSL Contexts

Novo na versão 3.2.

An SSL context holds various data longer-lived than single SSL connections, such as SSL configuration options, certificate(s) and private key(s). It also manages a cache of SSL sessions for server-side sockets, in order to speed up repeated connections from the same clients.

class ssl.**SSLContext**(*protocol=PROTOCOL_TLS*)

Create a new SSL context. You may pass *protocol* which must be one of the PROTOCOL_* constants defined in this module. The parameter specifies which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server's choice. Most of the versions are not interoperable with the other versions. If not specified, the default is *PROTOCOL_TLS*; it provides the most compatibility with other versions.

Here's a table showing which versions in a client (down the side) can connect to which versions in a server (along the top):

client / **server**	**SSLv2**	**SSLv3**	**TLS**[3]	**TLSv1**	**TLSv1.1**	**TLSv1.2**
SSLv2	sim	não	no[1]	não	não	não
SSLv3	não	sim	no[2]	não	não	não
TLS (SSLv23)[3]	no[1]	no[2]	sim	sim	sim	sim
TLSv1	não	não	sim	sim	não	não
TLSv1.1	não	não	sim	não	sim	não
TLSv1.2	não	não	sim	não	não	sim

Ver também:

create_default_context() lets the *ssl* module choose security settings for a given purpose.

Alterado na versão 3.6: The context is created with secure default values. The options *OP_NO_COMPRESSION*, *OP_CIPHER_SERVER_PREFERENCE*, *OP_SINGLE_DH_USE*, *OP_SINGLE_ECDH_USE*, *OP_NO_SSLv2* (except for *PROTOCOL_SSLv2*), and *OP_NO_SSLv3* (except for *PROTOCOL_SSLv3*) are set by default. The initial cipher suite list contains only HIGH ciphers, no NULL ciphers and no MD5 ciphers (except for *PROTOCOL_SSLv2*).

SSLContext objects have the following methods and attributes:

SSLContext.**cert_store_stats**()

Get statistics about quantities of loaded X.509 certificates, count of X.509 certificates flagged as CA certificates and certificate revocation lists as dictionary.

Example for a context with one CA cert and one other cert:

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

Novo na versão 3.4.

SSLContext.**load_cert_chain**(*certfile*, *keyfile=None*, *password=None*)

Load a private key and the corresponding certificate. The *certfile* string must be the path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate's authenticity. The *keyfile* string, if present, must point to a file containing the private key in. Otherwise the private key will be taken from *certfile* as well. See the discussion of *Certificados* for more information on how the certificate is stored in the *certfile*.

[3] TLS 1.3 protocol will be available with *PROTOCOL_TLS* in OpenSSL >= 1.1.1. There is no dedicated PROTOCOL constant for just TLS 1.3.
[1] *SSLContext* disables SSLv2 with *OP_NO_SSLv2* by default.
[2] *SSLContext* disables SSLv3 with *OP_NO_SSLv3* by default.

The *password* argument may be a function to call to get the password for decrypting the private key. It will only be called if the private key is encrypted and a password is necessary. It will be called with no arguments, and it should return a string, bytes, or bytearray. If the return value is a string it will be encoded as UTF-8 before using it to decrypt the key. Alternatively a string, bytes, or bytearray value may be supplied directly as the *password* argument. It will be ignored if the private key is not encrypted and no password is needed.

If the *password* argument is not specified and a password is required, OpenSSL's built-in password prompting mechanism will be used to interactively prompt the user for a password.

An *SSLError* is raised if the private key doesn't match with the certificate.

Alterado na versão 3.3: New optional argument *password*.

SSLContext.**load_default_certs**(*purpose=Purpose.SERVER_AUTH*)

Load a set of default "certification authority" (CA) certificates from default locations. On Windows it loads CA certs from the `CA` and `ROOT` system stores. On other systems it calls *SSLContext.set_default_verify_paths()*. In the future the method may load CA certificates from other locations, too.

The *purpose* flag specifies what kind of CA certificates are loaded. The default settings *Purpose.SERVER_AUTH* loads certificates, that are flagged and trusted for TLS web server authentication (client side sockets). *Purpose.CLIENT_AUTH* loads CA certificates for client certificate verification on the server side.

Novo na versão 3.4.

SSLContext.**load_verify_locations**(*cafile=None*, *capath=None*, *cadata=None*)

Load a set of "certification authority" (CA) certificates used to validate other peers' certificates when *verify_mode* is other than *CERT_NONE*. At least one of *cafile* or *capath* must be specified.

This method can also load certification revocation lists (CRLs) in PEM or DER format. In order to make use of CRLs, *SSLContext.verify_flags* must be configured properly.

The *cafile* string, if present, is the path to a file of concatenated CA certificates in PEM format. See the discussion of *Certificados* for more information about how to arrange the certificates in this file.

The *capath* string, if present, is the path to a directory containing several CA certificates in PEM format, following an OpenSSL specific layout.

The *cadata* object, if present, is either an ASCII string of one or more PEM-encoded certificates or a *bytes-like object* of DER-encoded certificates. Like with *capath* extra lines around PEM-encoded certificates are ignored but at least one certificate must be present.

Alterado na versão 3.4: New optional argument *cadata*

SSLContext.**get_ca_certs**(*binary_form=False*)

Get a list of loaded "certification authority" (CA) certificates. If the `binary_form` parameter is *False* each list entry is a dict like the output of *SSLSocket.getpeercert()*. Otherwise the method returns a list of DER-encoded certificates. The returned list does not contain certificates from *capath* unless a certificate was requested and loaded by a SSL connection.

Nota: Certificates in a capath directory aren't loaded unless they have been used at least once.

Novo na versão 3.4.

SSLContext.**get_ciphers**()

Get a list of enabled ciphers. The list is in order of cipher priority. See *SSLContext.set_ciphers()*.

Exemplo:

```
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDSA')
>>> ctx.get_ciphers()  # OpenSSL 1.0.x
[{'alg_bits': 256,
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'id': 50380848,
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 256},
 {'alg_bits': 128,
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'id': 50380847,
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 128}]
```

On OpenSSL 1.1 and newer the cipher dict contains additional fields:

```
>>> ctx.get_ciphers()  # OpenSSL 1.1+
[{'aead': True,
  'alg_bits': 256,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'digest': None,
  'id': 50380848,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1.2',
  'strength_bits': 256,
  'symmetric': 'aes-256-gcm'},
 {'aead': True,
  'alg_bits': 128,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'digest': None,
  'id': 50380847,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1.2',
  'strength_bits': 128,
  'symmetric': 'aes-128-gcm'}]
```

Availability: OpenSSL 1.0.2+.

Novo na versão 3.6.

SSLContext.**set_default_verify_paths**()
: Load a set of default "certification authority" (CA) certificates from a filesystem path defined when building the OpenSSL library. Unfortunately, there's no easy way to know whether this method succeeds: no error is returned if no certificates are to be found. When the OpenSSL library is provided as part of the operating system, though, it is likely to be configured properly.

SSLContext.**set_ciphers**(*ciphers*)
: Set the available ciphers for sockets created with this context. It should be a string in the OpenSSL cipher list

format. If no cipher can be selected (because compile-time options or other configuration forbids use of all the specified ciphers), an *SSLError* will be raised.

Nota: when connected, the *SSLSocket.cipher()* method of SSL sockets will give the currently selected cipher.

OpenSSL 1.1.1 has TLS 1.3 cipher suites enabled by default. The suites cannot be disabled with *set_ciphers()*.

SSLContext.**set_alpn_protocols**(*protocols*)

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of ASCII strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to **RFC 7301**. After a successful handshake, the *SSLSocket.selected_alpn_protocol()* method will return the agreed-upon protocol.

This method will raise *NotImplementedError* if *HAS_ALPN* is `False`.

OpenSSL 1.1.0 to 1.1.0e will abort the handshake and raise *SSLError* when both sides support ALPN but cannot agree on a protocol. 1.1.0f+ behaves like 1.0.2, *SSLSocket.selected_alpn_protocol()* returns None.

Novo na versão 3.5.

SSLContext.**set_npn_protocols**(*protocols*)

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to the Application Layer Protocol Negotiation. After a successful handshake, the *SSLSocket.selected_npn_protocol()* method will return the agreed-upon protocol.

This method will raise *NotImplementedError* if *HAS_NPN* is `False`.

Novo na versão 3.3.

SSLContext.**sni_callback**

Register a callback function that will be called after the TLS Client Hello handshake message has been received by the SSL/TLS server when the TLS client specifies a server name indication. The server name indication mechanism is specified in **RFC 6066** section 3 - Server Name Indication.

Only one callback can be set per `SSLContext`. If *sni_callback* is set to `None` then the callback is disabled. Calling this function a subsequent time will disable the previously registered callback.

The callback function will be called with three arguments; the first being the *ssl.SSLSocket*, the second is a string that represents the server name that the client is intending to communicate (or *None* if the TLS Client Hello does not contain a server name) and the third argument is the original *SSLContext*. The server name argument is text. For internationalized domain name, the server name is an IDN A-label (`"xn--pythn-mua.org"`).

A typical use of this callback is to change the *ssl.SSLSocket*'s *SSLSocket.context* attribute to a new object of type *SSLContext* representing a certificate chain that matches the server name.

Due to the early negotiation phase of the TLS connection, only limited methods and attributes are usable like *SSLSocket.selected_alpn_protocol()* and *SSLSocket.context*. *SSLSocket.getpeercert()*, *SSLSocket.getpeercert()*, *SSLSocket.cipher()* and `SSLSocket.compress()` methods require that the TLS connection has progressed beyond the TLS Client Hello and therefore will not contain return meaningful values nor can they be called safely.

The *sni_callback* function must return `None` to allow the TLS negotiation to continue. If a TLS failure is required, a constant *ALERT_DESCRIPTION_** can be returned. Other return values will result in a TLS fatal error with *ALERT_DESCRIPTION_INTERNAL_ERROR*.

If an exception is raised from the *sni_callback* function the TLS connection will terminate with a fatal TLS alert message *ALERT_DESCRIPTION_HANDSHAKE_FAILURE*.

This method will raise *NotImplementedError* if the OpenSSL library had OPENSSL_NO_TLSEXT defined when it was built.

Novo na versão 3.7.

SSLContext.**set_servername_callback**(*server_name_callback*)
This is a legacy API retained for backwards compatibility. When possible, you should use *sni_callback* instead. The given *server_name_callback* is similar to *sni_callback*, except that when the server hostname is an IDN-encoded internationalized domain name, the *server_name_callback* receives a decoded U-label (`"pythön.org"`).

If there is an decoding error on the server name, the TLS connection will terminate with an *ALERT_DESCRIPTION_INTERNAL_ERROR* fatal TLS alert message to the client.

Novo na versão 3.4.

SSLContext.**load_dh_params**(*dhfile*)
Load the key generation parameters for Diffie-Hellman (DH) key exchange. Using DH key exchange improves forward secrecy at the expense of computational resources (both on the server and on the client). The *dhfile* parameter should be the path to a file containing DH parameters in PEM format.

This setting doesn't apply to client sockets. You can also use the *OP_SINGLE_DH_USE* option to further improve security.

Novo na versão 3.3.

SSLContext.**set_ecdh_curve**(*curve_name*)
Set the curve name for Elliptic Curve-based Diffie-Hellman (ECDH) key exchange. ECDH is significantly faster than regular DH while arguably as secure. The *curve_name* parameter should be a string describing a well-known elliptic curve, for example `prime256v1` for a widely supported curve.

This setting doesn't apply to client sockets. You can also use the *OP_SINGLE_ECDH_USE* option to further improve security.

This method is not available if *HAS_ECDH* is `False`.

Novo na versão 3.3.

Ver também:

SSL/TLS & Perfect Forward Secrecy Vincent Bernat.

SSLContext.**wrap_socket**(*sock*, *server_side=False*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *server_hostname=None*, *session=None*)
Wrap an existing Python socket *sock* and return an instance of *SSLContext.sslsocket_class* (default *SSLSocket*). The returned SSL socket is tied to the context, its settings and certificates. *sock* must be a *SOCK_STREAM* socket; other socket types are unsupported.

The parameter `server_side` is a boolean which identifies whether server-side or client-side behavior is desired from this socket.

For client-side sockets, the context construction is lazy; if the underlying socket isn't connected yet, the context construction will be performed after `connect()` is called on the socket. For server-side sockets, if the socket has no remote peer, it is assumed to be a listening socket, and the server-side SSL wrapping is automatically performed on client connections accepted via the `accept()` method. The method may raise *SSLError*.

On client connections, the optional parameter *server_hostname* specifies the hostname of the service which we are connecting to. This allows a single server to host multiple SSL-based services with distinct certificates, quite similarly to HTTP virtual hosts. Specifying *server_hostname* will raise a *ValueError* if *server_side* is true.

The parameter `do_handshake_on_connect` specifies whether to do the SSL handshake automatically after doing a `socket.connect()`, or whether the application program will call it explicitly, by invoking the

SSLSocket.do_handshake() method. Calling *SSLSocket.do_handshake()* explicitly gives the program control over the blocking behavior of the socket I/O involved in the handshake.

The parameter `suppress_ragged_eofs` specifies how the `SSLSocket.recv()` method should signal unexpected EOF from the other end of the connection. If specified as *True* (the default), it returns a normal EOF (an empty bytes object) in response to unexpected EOF errors raised from the underlying socket; if *False*, it will raise the exceptions back to the caller.

session, see *session*.

Alterado na versão 3.5: Always allow a server_hostname to be passed, even if OpenSSL does not have SNI.

Alterado na versão 3.6: o argumento *session* foi adicionado.

Alterado na versão 3.7: The method returns on instance of *SSLContext.sslsocket_class* instead of hard-coded *SSLSocket*.

`SSLContext.`**`sslsocket_class`**

The return type of *SSLContext.wrap_socket()*, defaults to *SSLSocket*. The attribute can be overridden on instance of class in order to return a custom subclass of *SSLSocket*.

Novo na versão 3.7.

`SSLContext.`**`wrap_bio`**(*incoming*, *outgoing*, *server_side=False*, *server_hostname=None*, *session=None*)

Wrap the BIO objects *incoming* and *outgoing* and return an instance of *SSLContext.sslobject_class* (default *SSLObject*). The SSL routines will read input data from the incoming BIO and write data to the outgoing BIO.

The *server_side*, *server_hostname* and *session* parameters have the same meaning as in *SSLContext.wrap_socket()*.

Alterado na versão 3.6: o argumento *session* foi adicionado.

Alterado na versão 3.7: The method returns on instance of *SSLContext.sslobject_class* instead of hard-coded *SSLObject*.

`SSLContext.`**`sslobject_class`**

The return type of *SSLContext.wrap_bio()*, defaults to *SSLObject*. The attribute can be overridden on instance of class in order to return a custom subclass of *SSLObject*.

Novo na versão 3.7.

`SSLContext.`**`session_stats`**`()`

Get statistics about the SSL sessions created or managed by this context. A dictionary is returned which maps the names of each piece of information to their numeric values. For example, here is the total number of hits and misses in the session cache since the context was created:

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

`SSLContext.`**`check_hostname`**

Whether to match the peer cert's hostname with *match_hostname()* in *SSLSocket.do_handshake()*. The context's *verify_mode* must be set to *CERT_OPTIONAL* or *CERT_REQUIRED*, and you must pass *server_hostname* to *wrap_socket()* in order to match the hostname. Enabling hostname checking automatically sets *verify_mode* from *CERT_NONE* to *CERT_REQUIRED*. It cannot be set back to *CERT_NONE* as long as hostname checking is enabled. The *PROTOCOL_TLS_CLIENT* protocol enables hostname checking by default. With other protocols, hostname checking must be enabled explicitly.

Exemplo:

```
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

Novo na versão 3.4.

Alterado na versão 3.7: *verify_mode* is now automatically changed to *CERT_REQUIRED* when hostname checking is enabled and *verify_mode* is *CERT_NONE*. Previously the same operation would have failed with a *ValueError*.

Nota: This features requires OpenSSL 0.9.8f or newer.

SSLContext.**maximum_version**

A *TLSVersion* enum member representing the highest supported TLS version. The value defaults to *TLSVersion.MAXIMUM_SUPPORTED*. The attribute is read-only for protocols other than *PROTOCOL_TLS*, *PROTOCOL_TLS_CLIENT*, and *PROTOCOL_TLS_SERVER*.

The attributes *maximum_version*, *minimum_version* and *SSLContext.options* all affect the supported SSL and TLS versions of the context. The implementation does not prevent invalid combination. For example a context with *OP_NO_TLSv1_2* in *options* and *maximum_version* set to *TLSVersion.TLSv1_2* will not be able to establish a TLS 1.2 connection.

Nota: This attribute is not available unless the ssl module is compiled with OpenSSL 1.1.0g or newer.

Novo na versão 3.7.

SSLContext.**minimum_version**

Like *SSLContext.maximum_version* except it is the lowest supported version or *TLSVersion.MINIMUM_SUPPORTED*.

Nota: This attribute is not available unless the ssl module is compiled with OpenSSL 1.1.0g or newer.

Novo na versão 3.7.

SSLContext.**options**

An integer representing the set of SSL options enabled on this context. The default value is *OP_ALL*, but you can specify other options such as *OP_NO_SSLv2* by ORing them together.

Nota: With versions of OpenSSL older than 0.9.8m, it is only possible to set options, not to clear them. Attempting to clear an option (by resetting the corresponding bits) will raise a *ValueError*.

Alterado na versão 3.6: *SSLContext.options* returns *Options* flags:

```
>>> ssl.create_default_context().options
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION: 2197947391>
```

SSLContext.**post_handshake_auth**
Enable TLS 1.3 post-handshake client authentication. Post-handshake auth is disabled by default and a server can only request a TLS client certificate during the initial handshake. When enabled, a server may request a TLS client certificate at any time after the handshake.

When enabled on client-side sockets, the client signals the server that it supports post-handshake authentication.

When enabled on server-side sockets, *SSLContext.verify_mode* must be set to *CERT_OPTIONAL* or *CERT_REQUIRED*, too. The actual client cert exchange is delayed until *SSLSocket.verify_client_post_handshake()* is called and some I/O is performed.

Nota: Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the property value is None and can't be modified

Novo na versão 3.7.1.

SSLContext.**protocol**
The protocol version chosen when constructing the context. This attribute is read-only.

SSLContext.**hostname_checks_common_name**
Whether *check_hostname* falls back to verify the cert's subject common name in the absence of a subject alternative name extension (default: true).

Nota: Only writeable with OpenSSL 1.1.0 or higher.

Novo na versão 3.7.

SSLContext.**verify_flags**
The flags for certificate verification operations. You can set flags like *VERIFY_CRL_CHECK_LEAF* by ORing them together. By default OpenSSL does neither require nor verify certificate revocation lists (CRLs). Available only with openssl version 0.9.8+.

Novo na versão 3.4.

Alterado na versão 3.6: *SSLContext.verify_flags* returns *VerifyFlags* flags:

```
>>> ssl.create_default_context().verify_flags
<VerifyFlags.VERIFY_X509_TRUSTED_FIRST: 32768>
```

SSLContext.**verify_mode**
Whether to try to verify other peers' certificates and how to behave if verification fails. This attribute must be one of *CERT_NONE*, *CERT_OPTIONAL* or *CERT_REQUIRED*.

Alterado na versão 3.6: *SSLContext.verify_mode* returns *VerifyMode* enum:

```
>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```

19.3.4 Certificados

Certificates in general are part of a public-key / private-key system. In this system, each *principal*, (which may be a machine, or a person, or an organization) is assigned a unique two-part encryption key. One part of the key is public, and is called the *public key*; the other part is kept secret, and is called the *private key*. The two parts are related, in that if you encrypt a message with one of the parts, you can decrypt it with the other part, and **only** with the other part.

A certificate contains information about two principals. It contains the name of a *subject*, and the subject's public key. It also contains a statement by a second principal, the *issuer*, that the subject is who they claim to be, and that this is indeed the subject's public key. The issuer's statement is signed with the issuer's private key, which only the issuer knows. However, anyone can verify the issuer's statement by finding the issuer's public key, decrypting the statement with it, and comparing it to the other information in the certificate. The certificate also contains information about the time period over which it is valid. This is expressed as two fields, called "notBefore" and "notAfter".

In the Python use of certificates, a client or server can use a certificate to prove who they are. The other side of a network connection can also be required to produce a certificate, and that certificate can be validated to the satisfaction of the client or server that requires such validation. The connection attempt can be set to raise an exception if the validation fails. Validation is done automatically, by the underlying OpenSSL framework; the application need not concern itself with its mechanics. But the application does usually need to provide sets of certificates to allow this process to take place.

Python uses files to contain certificates. They should be formatted as "PEM" (see **RFC 1422**), which is a base-64 encoded form wrapped with a header line and a footer line:

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Certificate chains

The Python files which contain certificates can contain a sequence of certificates, sometimes called a *certificate chain*. This chain should start with the specific certificate for the principal who "is" the client or server, and then the certificate for the issuer of that certificate, and then the certificate for the issuer of *that* certificate, and so on up the chain till you get to a certificate which is *self-signed*, that is, a certificate which has the same subject and issuer, sometimes called a *root certificate*. The certificates should just be concatenated together in the certificate file. For example, suppose we had a three certificate chain, from our server certificate to the certificate of the certification authority that signed our server certificate, to the root certificate of the agency which issued the certification authority's certificate:

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

CA certificates

If you are going to require validation of the other side of the connection's certificate, you need to provide a "CA certs" file, filled with the certificate chains for each issuer you are willing to trust. Again, this file just contains these chains concatenated together. For validation, Python will use the first chain it finds in the file which matches. The platform's certificates file can be used by calling *SSLContext.load_default_certs()*, this is done automatically with *create_default_context()*.

Combined key and certificate

Often the private key is stored in the same file as the certificate; in this case, only the `certfile` parameter to *SSLContext.load_cert_chain()* and *wrap_socket()* needs to be passed. If the private key is stored with the certificate, it should come before the first certificate in the certificate chain:

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Self-signed certificates

If you are going to create a server that provides SSL-encrypted connection services, you will need to acquire a certificate for that service. There are many ways of acquiring appropriate certificates, such as buying one from a certification authority. Another common practice is to generate a self-signed certificate. The simplest way to do this is with the OpenSSL package, using something like the following:

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
.......++++++
.............................++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

The disadvantage of a self-signed certificate is that it is its own root certificate, and no one else will have it in their cache of known (and trusted) root certificates.

19.3.5 Exemplos

Testing for SSL support

To test for the presence of SSL support in a Python installation, user code should use the following idiom:

```
try:
    import ssl
except ImportError:
    pass
else:
    ...  # do something that requires SSL support
```

Client-side operation

This example creates a SSL context with the recommended security settings for client sockets, including automatic certificate verification:

```
>>> context = ssl.create_default_context()
```

If you prefer to tune security settings yourself, you might create a context from scratch (but beware that you might not get the settings right):

```
>>> context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(this snippet assumes your operating system places a bundle of all CA certificates in `/etc/ssl/certs/ca-bundle.crt`; if not, you'll get an error and have to adjust the location)

The *PROTOCOL_TLS_CLIENT* protocol configures the context for cert validation and hostname verification. *verify_mode* is set to *CERT_REQUIRED* and *check_hostname* is set to `True`. All other protocols create SSL contexts with insecure defaults.

When you use the context to connect to a server, *CERT_REQUIRED* and *check_hostname* validate the server certificate: it ensures that the server certificate was signed with one of the CA certificates, checks the signature for correctness, and verifies other properties like validity and identity of the hostname:

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                            server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

You may then fetch the certificate:

```
>>> cert = conn.getpeercert()
```

Visual inspection shows that the certificate does identify the desired service (that is, the HTTPS host `www.python.org`):

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServerCA.crt
↪',),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': ((('countryName', 'US'),),
```

(continua na próxima página)

(continuação da página anterior)

```
           (('organizationName', 'DigiCert Inc'),),
           (('organizationalUnitName', 'www.digicert.com'),),
           (('commonName', 'DigiCert SHA2 Extended Validation Server CA'),)),
 'notAfter': 'Sep  9 12:00:00 2016 GMT',
 'notBefore': 'Sep  5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': ((('businessCategory', 'Private Organization'),),
             (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
             (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
             (('serialNumber', '3359300'),),
             (('streetAddress', '16 Allen Rd'),),
             (('postalCode', '03894-4801'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'NH'),),
             (('localityName', 'Wolfeboro'),),
             (('organizationName', 'Python Software Foundation'),),
             (('commonName', 'www.python.org'),)),
 'subjectAltName': (('DNS', 'www.python.org'),
                    ('DNS', 'python.org'),
                    ('DNS', 'pypi.org'),
                    ('DNS', 'docs.python.org'),
                    ('DNS', 'testpypi.org'),
                    ('DNS', 'bugs.python.org'),
                    ('DNS', 'wiki.python.org'),
                    ('DNS', 'hg.python.org'),
                    ('DNS', 'mail.python.org'),
                    ('DNS', 'packaging.python.org'),
                    ('DNS', 'pythonhosted.org'),
                    ('DNS', 'www.pythonhosted.org'),
                    ('DNS', 'test.pythonhosted.org'),
                    ('DNS', 'us.pycon.org'),
                    ('DNS', 'id.python.org')),
 'version': 3}
```

Now the SSL channel is established and the certificate verified, you can proceed to talk with the server:

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
 b'Connection: close',
 b'',
 b'']
```

See the discussion of *Considerações de segurança* below.

Server-side operation

For server operation, typically you'll need to have a server certificate, and private key, each in a file. You'll first create a context holding the key and the certificate, so that clients can check your authenticity. Then you'll open a socket, bind it to a port, call `listen()` on it, and start waiting for clients to connect:

```
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.mydomain.com', 10023))
bindsocket.listen(5)
```

When a client connects, you'll call `accept()` on the socket to get the new socket from the other end, and use the context's *SSLContext.wrap_socket()* method to create a server-side SSL socket for the connection:

```
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

Then you'll read data from the `connstream` and do something with it till you are finished with the client (or the client is finished with you):

```
def deal_with_client(connstream):
    data = connstream.recv(1024)
    # empty data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.recv(1024)
    # finished with client
```

And go back to listening for new client connections (of course, a real server would probably handle each client connection in a separate thread, or put the sockets in *non-blocking mode* and use an event loop).

19.3.6 Notes on non-blocking sockets

SSL sockets behave slightly different than regular sockets in non-blocking mode. When working with non-blocking sockets, there are thus several things you need to be aware of:

- Most *SSLSocket* methods will raise either *SSLWantWriteError* or *SSLWantReadError* instead of *BlockingIOError* if an I/O operation would block. *SSLWantReadError* will be raised if a read operation on the underlying socket is necessary, and *SSLWantWriteError* for a write operation on the underlying socket. Note that attempts to *write* to an SSL socket may require *reading* from the underlying socket first, and attempts to *read* from the SSL socket may require a prior *write* to the underlying socket.

  Alterado na versão 3.5: In earlier Python versions, the `SSLSocket.send()` method returned zero instead of raising *SSLWantWriteError* or *SSLWantReadError*.

- Calling *select()* tells you that the OS-level socket can be read from (or written to), but it does not imply that there is sufficient data at the upper SSL layer. For example, only part of an SSL frame might have arrived. Therefore, you must be ready to handle `SSLSocket.recv()` and `SSLSocket.send()` failures, and retry after another call to *select()*.
- Conversely, since the SSL layer has its own framing, a SSL socket may still have data available for reading without *select()* being aware of it. Therefore, you should first call `SSLSocket.recv()` to drain any potentially available data, and then only block on a *select()* call if still necessary.

  (of course, similar provisions apply when using other primitives such as *poll()*, or those in the *selectors* module)
- The SSL handshake itself will be non-blocking: the *SSLSocket.do_handshake()* method has to be retried until it returns successfully. Here is a synopsis using *select()* to wait for the socket's readiness:

```
while True:
    try:
        sock.do_handshake()
        break
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```

Ver também:

The *asyncio* module supports *non-blocking SSL sockets* and provides a higher level API. It polls for events using the *selectors* module and handles *SSLWantWriteError*, *SSLWantReadError* and *BlockingIOError* exceptions. It runs the SSL handshake asynchronously as well.

19.3.7 Memory BIO Support

Novo na versão 3.5.

Ever since the SSL module was introduced in Python 2.6, the *SSLSocket* class has provided two related but distinct areas of functionality:

- SSL protocol handling
- Network IO

The network IO API is identical to that provided by *socket.socket*, from which *SSLSocket* also inherits. This allows an SSL socket to be used as a drop-in replacement for a regular socket, making it very easy to add SSL support to an existing application.

Combining SSL protocol handling and network IO usually works well, but there are some cases where it doesn't. An example is async IO frameworks that want to use a different IO multiplexing model than the "select/poll on a file descriptor" (readiness based) model that is assumed by *socket.socket* and by the internal OpenSSL socket IO routines. This is mostly relevant for platforms like Windows where this model is not efficient. For this purpose, a reduced scope variant of *SSLSocket* called *SSLObject* is provided.

class ssl.**SSLObject**

A reduced-scope variant of *SSLSocket* representing an SSL protocol instance that does not contain any network IO methods. This class is typically used by framework authors that want to implement asynchronous IO for SSL through memory buffers.

This class implements an interface on top of a low-level SSL object as implemented by OpenSSL. This object captures the state of an SSL connection but does not provide any network IO itself. IO needs to be performed through separate "BIO" objects which are OpenSSL's IO abstraction layer.

This class has no public constructor. An *SSLObject* instance must be created using the *wrap_bio()* method. This method will create the *SSLObject* instance and bind it to a pair of BIOs. The *incoming* BIO is used to pass data from Python to the SSL protocol instance, while the *outgoing* BIO is used to pass data the other way around.

The following methods are available:

- *context*
- *server_side*
- *server_hostname*
- *session*
- *session_reused*
- *read()*
- *write()*
- *getpeercert()*
- *selected_npn_protocol()*
- *cipher()*
- *shared_ciphers()*
- *compression()*
- *pending()*
- *do_handshake()*
- *unwrap()*
- *get_channel_binding()*

When compared to *SSLSocket*, this object lacks the following features:

- Any form of network IO; `recv()` and `send()` read and write only to the underlying *MemoryBIO* buffers.
- There is no *do_handshake_on_connect* machinery. You must always manually call *do_handshake()* to start the handshake.
- There is no handling of *suppress_ragged_eofs*. All end-of-file conditions that are in violation of the protocol are reported via the *SSLEOFError* exception.
- The method *unwrap()* call does not return anything, unlike for an SSL socket where it returns the underlying socket.
- The *server_name_callback* callback passed to *SSLContext.set_servername_callback()* will get an *SSLObject* instance instead of a *SSLSocket* instance as its first parameter.

Some notes related to the use of *SSLObject*:

- All IO on an *SSLObject* is *non-blocking*. This means that for example *read()* will raise an *SSLWantReadError* if it needs more data than the incoming BIO has available.
- There is no module-level `wrap_bio()` call like there is for *wrap_socket()*. An *SSLObject* is always created via an *SSLContext*.

Alterado na versão 3.7: *SSLObject* instances must to created with *wrap_bio()*. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

An SSLObject communicates with the outside world using memory buffers. The class *MemoryBIO* provides a memory buffer that can be used for this purpose. It wraps an OpenSSL memory BIO (Basic IO) object:

class ssl.**MemoryBIO**
A memory buffer that can be used to pass data between Python and an SSL protocol instance.

pending
Return the number of bytes currently in the memory buffer.

eof
A boolean indicating whether the memory BIO is current at the end-of-file position.

read(*n=-1*)
Read up to *n* bytes from the memory buffer. If *n* is not specified or negative, all bytes are returned.

write(*buf*)
Write the bytes from *buf* to the memory BIO. The *buf* argument must be an object supporting the buffer protocol.

The return value is the number of bytes written, which is always equal to the length of *buf*.

write_eof()
Write an EOF marker to the memory BIO. After this method has been called, it is illegal to call *write()*. The attribute *eof* will become true after all data currently in the buffer has been read.

19.3.8 SSL session

Novo na versão 3.6.

class ssl.**SSLSession**
Session object used by *session*.

id

time

timeout

ticket_lifetime_hint

has_ticket

19.3.9 Considerações de segurança

Best defaults

For **client use**, if you don't have any special requirements for your security policy, it is highly recommended that you use the *create_default_context()* function to create your SSL context. It will load the system's trusted CA certificates, enable certificate validation and hostname checking, and try to choose reasonably secure protocol and cipher settings.

For example, here is how you would use the *smtplib.SMTP* class to create a trusted, secure connection to a SMTP server:

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

If a client certificate is needed for the connection, it can be added with *SSLContext.load_cert_chain()*.

By contrast, if you create the SSL context by calling the *SSLContext* constructor yourself, it will not have certificate validation nor hostname checking enabled by default. If you do so, please read the paragraphs below to achieve a good security level.

Manual settings

Verifying certificates

When calling the *SSLContext* constructor directly, *CERT_NONE* is the default. Since it does not authenticate the other peer, it can be insecure, especially in client mode where most of time you would like to ensure the authenticity of the server you're talking to. Therefore, when in client mode, it is highly recommended to use *CERT_REQUIRED*. However, it is in itself not sufficient; you also have to check that the server certificate, which can be obtained by calling *SSLSocket.getpeercert()*, matches the desired service. For many protocols and applications, the service can be identified by the hostname; in this case, the *match_hostname()* function can be used. This common check is automatically performed when *SSLContext.check_hostname* is enabled.

Alterado na versão 3.7: Hostname matchings is now performed by OpenSSL. Python no longer uses *match_hostname()*.

In server mode, if you want to authenticate your clients using the SSL layer (rather than using a higher-level authentication mechanism), you'll also have to specify *CERT_REQUIRED* and similarly check the client certificate.

Protocol versions

SSL versions 2 and 3 are considered insecure and are therefore dangerous to use. If you want maximum compatibility between clients and servers, it is recommended to use *PROTOCOL_TLS_CLIENT* or *PROTOCOL_TLS_SERVER* as the protocol version. SSLv2 and SSLv3 are disabled by default.

```
>>> client_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> client_context.options |= ssl.OP_NO_TLSv1
>>> client_context.options |= ssl.OP_NO_TLSv1_1
```

The SSL context created above will only allow TLSv1.2 and later (if supported by your system) connections to a server. *PROTOCOL_TLS_CLIENT* implies certificate validation and hostname checks by default. You have to load certificates into the context.

Cipher selection

If you have advanced security requirements, fine-tuning of the ciphers enabled when negotiating a SSL session is possible through the *SSLContext.set_ciphers()* method. Starting from Python 3.2.3, the ssl module disables certain weak ciphers by default, but you may want to further restrict the cipher choice. Be sure to read OpenSSL's documentation about the cipher list format. If you want to check which ciphers are enabled by a given cipher list, use *SSLContext.get_ciphers()* or the `openssl ciphers` command on your system.

Multi-processing

If using this module as part of a multi-processed application (using, for example the *multiprocessing* or *concurrent.futures* modules), be aware that OpenSSL's internal random number generator does not properly handle forked processes. Applications must change the PRNG state of the parent process if they use any SSL feature with *os.fork()*. Any successful call of *RAND_add()*, *RAND_bytes()* or *RAND_pseudo_bytes()* is sufficient.

19.3.10 TLS 1.3

Novo na versão 3.7.

Python has provisional and experimental support for TLS 1.3 with OpenSSL 1.1.1. The new protocol behaves slightly differently than previous version of TLS/SSL. Some new TLS 1.3 features are not yet available.

- TLS 1.3 uses a disjunct set of cipher suites. All AES-GCM and ChaCha20 cipher suites are enabled by default. The method *SSLContext.set_ciphers()* cannot enable or disable any TLS 1.3 ciphers yet, but *SSLContext.get_ciphers()* returns them.
- Session tickets are no longer sent as part of the initial handshake and are handled differently. *SSLSocket.session* and *SSLSession* are not compatible with TLS 1.3.
- Client-side certificates are also no longer verified during the initial handshake. A server can request a certificate at any time. Clients process certificate requests while they send or receive application data from the server.
- TLS 1.3 features like early data, deferred TLS client cert request, signature algorithm configuration, and rekeying are not supported yet.

19.3.11 LibreSSL support

LibreSSL is a fork of OpenSSL 1.0.1. The ssl module has limited support for LibreSSL. Some features are not available when the ssl module is compiled with LibreSSL.

- LibreSSL >= 2.6.1 no longer supports NPN. The methods *SSLContext.set_npn_protocols()* and *SSLSocket.selected_npn_protocol()* are not available.
- *SSLContext.set_default_verify_paths()* ignores the env vars `SSL_CERT_FILE` and `SSL_CERT_PATH` although *get_default_verify_paths()* still reports them.

Ver também:

Class *socket.socket* Documentation of underlying *socket* class

SSL/TLS Strong Encryption: An Introduction Intro from the Apache HTTP Server documentation

RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management Steve Kent

RFC 4086: Randomness Requirements for Security Donald E., Jeffrey I. Schiller

RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile D. Cooper

RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2 T. Dierks et. al.

RFC 6066: Transport Layer Security (TLS) Extensions D. Eastlake

IANA TLS: Transport Layer Security (TLS) Parameters IANA

RFC 7525: Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTL
IETF

Mozilla's Server Side TLS recommendations Mozilla

19.4 `select` — Waiting for I/O completion

This module provides access to the `select()` and `poll()` functions available in most operating systems, `devpoll()` available on Solaris and derivatives, `epoll()` available on Linux 2.5+ and `kqueue()` available on most BSD. Note that on Windows, it only works for sockets; on other operating systems, it also works for other file types (in particular, on Unix, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

Nota: The *`selectors`* module allows high-level and efficient I/O multiplexing, built upon the *`select`* module primitives. Users are encouraged to use the *`selectors`* module instead, unless they want precise control over the OS-level primitives used.

O módulo define o seguinte:

exception `select.`**error**
A deprecated alias of *`OSError`*.

Alterado na versão 3.3: Following **PEP 3151**, this class was made an alias of *`OSError`*.

`select.`**devpoll**`()`
(Only supported on Solaris and derivatives.) Returns a `/dev/poll` polling object; see section */dev/poll Polling Objects* below for the methods supported by devpoll objects.

`devpoll()` objects are linked to the number of file descriptors allowed at the time of instantiation. If your program reduces this value, `devpoll()` will fail. If your program increases this value, `devpoll()` may return an incomplete list of active file descriptors.

The new file descriptor is *non-inheritable*.

Novo na versão 3.3.

Alterado na versão 3.4: O novo descritor de arquivo agora é não-hereditário.

`select.`**epoll**`(`*sizehint=-1*, *flags=0*`)`
(Only supported on Linux 2.5.44 and newer.) Return an edge polling object, which can be used as Edge or Level Triggered interface for I/O events.

sizehint informs epoll about the expected number of events to be registered. It must be positive, or *-1* to use the default. It is only used on older systems where `epoll_create1()` is not available; otherwise it has no effect (though its value is still checked).

flags is deprecated and completely ignored. However, when supplied, its value must be `0` or `select.EPOLL_CLOEXEC`, otherwise `OSError` is raised.

See the *Edge and Level Trigger Polling (epoll) Objects* section below for the methods supported by epolling objects.

`epoll` objects support the context management protocol: when used in a `with` statement, the new file descriptor is automatically closed at the end of the block.

The new file descriptor is *non-inheritable*.

Alterado na versão 3.3: Added the *flags* parameter.

Alterado na versão 3.4: Support for the `with` statement was added. The new file descriptor is now non-inheritable.

Obsoleto desde a versão 3.4: The *flags* parameter. `select.EPOLL_CLOEXEC` is used by default now. Use `os.set_inheritable()` to make the file descriptor inheritable.

`select.poll()`
(Not supported by all operating systems.) Returns a polling object, which supports registering and unregistering file descriptors, and then polling them for I/O events; see section *Polling Objects* below for the methods supported by polling objects.

`select.kqueue()`
(Only supported on BSD.) Returns a kernel queue object; see section *Kqueue Objects* below for the methods supported by kqueue objects.

The new file descriptor is *non-inheritable*.

Alterado na versão 3.4: O novo descritor de arquivo agora é não-hereditário.

`select.kevent`(*ident*, *filter=KQ_FILTER_READ*, *flags=KQ_EV_ADD*, *fflags=0*, *data=0*, *udata=0*)
(Only supported on BSD.) Returns a kernel event object; see section *Kevent Objects* below for the methods supported by kevent objects.

`select.select`(*rlist*, *wlist*, *xlist*[, *timeout*])
This is a straightforward interface to the Unix `select()` system call. The first three arguments are iterables of 'waitable objects': either integers representing file descriptors or objects with a parameterless method named `fileno()` returning such an integer:

- *rlist*: wait until ready for reading
- *wlist*: wait until ready for writing
- *xlist*: wait for an "exceptional condition" (see the manual page for what your system considers such a condition)

Empty iterables are allowed, but acceptance of three empty iterables is platform-dependent. (It is known to work on Unix but not on Windows.) The optional *timeout* argument specifies a time-out as a floating point number in seconds. When the *timeout* argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Among the acceptable object types in the iterables are Python *file objects* (e.g. `sys.stdin`, or objects returned by `open()` or `os.popen()`), socket objects returned by `socket.socket()`. You may also define a *wrapper* class yourself, as long as it has an appropriate `fileno()` method (that really returns a file descriptor, not just a random integer).

Nota: File objects on Windows are not acceptable, but sockets are. On Windows, the underlying `select()` function is provided by the WinSock library, and does not handle file descriptors that don't originate from WinSock.

Alterado na versão 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising `InterruptedError`.

`select.PIPE_BUF`
The minimum number of bytes which can be written without blocking to a pipe when the pipe has been reported as ready for writing by `select()`, `poll()` or another interface in this module. This doesn't apply to other kind of file-like objects such as sockets.

This value is guaranteed by POSIX to be at least 512.

Disponibilidade: Unix

Novo na versão 3.2.

19.4.1 `/dev/poll` Polling Objects

Solaris and derivatives have `/dev/poll`. While `select()` is O(highest file descriptor) and `poll()` is O(number of file descriptors), `/dev/poll` is O(active file descriptors).

`/dev/poll` behaviour is very close to the standard `poll()` object.

`devpoll.close()`
Close the file descriptor of the polling object.

Novo na versão 3.4.

`devpoll.closed`
`True` if the polling object is closed.

Novo na versão 3.4.

`devpoll.fileno()`
Return the file descriptor number of the polling object.

Novo na versão 3.4.

`devpoll.register`(*fd*[, *eventmask*])
Register a file descriptor with the polling object. Future calls to the *poll()* method will then check whether the file descriptor has any pending I/O events. *fd* can be either an integer, or an object with a *fileno()* method that returns an integer. File objects implement `fileno()`, so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for. The constants are the same that with `poll()` object. The default value is a combination of the constants `POLLIN`, `POLLPRI`, and `POLLOUT`.

> **Aviso:** Registering a file descriptor that's already registered is not an error, but the result is undefined. The appropriate action is to unregister or modify it first. This is an important difference compared with `poll()`.

`devpoll.modify`(*fd*[, *eventmask*])
This method does an *unregister()* followed by a *register()*. It is (a bit) more efficient that doing the same explicitly.

`devpoll.unregister`(*fd*)
Remove a file descriptor being tracked by a polling object. Just like the *register()* method, *fd* can be an integer or an object with a *fileno()* method that returns an integer.

Attempting to remove a file descriptor that was never registered is safely ignored.

`devpoll.poll`([*timeout*])
Polls the set of registered file descriptors, and returns a possibly-empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with bits set for the reported events for that descriptor — `POLLIN` for waiting input, `POLLOUT` to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, -1, or *None*, the call will block until there is an event for this poll object.

Alterado na versão 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

19.4.2 Edge and Level Trigger Polling (epoll) Objects

https://linux.die.net/man/4/epoll

eventmask

Constante	Significado
EPOLLIN	Disponível para leitura
EPOLLOUT	Disponível para escrita
EPOLLPRI	Urgent data for read
EPOLLERR	Error condition happened on the assoc. fd
EPOLLHUP	Hang up happened on the assoc. fd
EPOLLET	Set Edge Trigger behavior, the default is Level Trigger behavior
EPOLLONESHOT	Set one-shot behavior. After one event is pulled out, the fd is internally disabled
EPOLLEXCLUSIVE	Wake only one epoll object when the associated fd has an event. The default (if this flag is not set) is to wake all epoll objects polling on a fd.
EPOLLRDHUP	Stream socket peer closed connection or shut down writing half of connection.
EPOLLRDNORM	Equivalent to EPOLLIN
EPOLLRDBAND	Priority data band can be read.
EPOLLWRNORM	Equivalente a EPOLLOUT
EPOLLWRBAND	Priority data may be written.
EPOLLMSG	Ignorado.

Novo na versão 3.6: EPOLLEXCLUSIVE was added. It's only supported by Linux Kernel 4.5 or later.

epoll.**close**()
Close the control file descriptor of the epoll object.

epoll.**closed**
True if the epoll object is closed.

epoll.**fileno**()
Return the file descriptor number of the control fd.

epoll.**fromfd**(*fd*)
Create an epoll object from a given file descriptor.

epoll.**register**(*fd*[, *eventmask*])
Register a fd descriptor with the epoll object.

epoll.**modify**(*fd*, *eventmask*)
Modify a registered file descriptor.

epoll.**unregister**(*fd*)
Remove a registered file descriptor from the epoll object.

epoll.**poll**(*timeout=-1*, *maxevents=-1*)
Wait for events. timeout in seconds (float)

Alterado na versão 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

19.4.3 Polling Objects

The `poll()` system call, supported on most Unix systems, provides better scalability for network servers that service many, many clients at the same time. `poll()` scales better because the system call only requires listing the file descriptors of interest, while `select()` builds a bitmap, turns on bits for the fds of interest, and then afterward the whole bitmap has to be linearly scanned again. `select()` is O(highest file descriptor), while `poll()` is O(number of file descriptors).

`poll.`**`register`**(*fd*[, *eventmask*])

Register a file descriptor with the polling object. Future calls to the *poll()* method will then check whether the file descriptor has any pending I/O events. *fd* can be either an integer, or an object with a *fileno()* method that returns an integer. File objects implement `fileno()`, so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for, and can be a combination of the constants `POLLIN`, `POLLPRI`, and `POLLOUT`, described in the table below. If not specified, the default value used will check for all 3 types of events.

Constante	Significado
`POLLIN`	There is data to read
`POLLPRI`	There is urgent data to read
`POLLOUT`	Ready for output: writing will not block
`POLLERR`	Error condition of some sort
`POLLHUP`	Hung up
`POLLRDHUP`	Stream socket peer closed connection, or shut down writing half of connection
`POLLNVAL`	Invalid request: descriptor not open

Registering a file descriptor that's already registered is not an error, and has the same effect as registering the descriptor exactly once.

`poll.`**`modify`**(*fd*, *eventmask*)

Modifies an already registered fd. This has the same effect as `register(fd, eventmask)`. Attempting to modify a file descriptor that was never registered causes an *OSError* exception with errno `ENOENT` to be raised.

`poll.`**`unregister`**(*fd*)

Remove a file descriptor being tracked by a polling object. Just like the *register()* method, *fd* can be an integer or an object with a *fileno()* method that returns an integer.

Attempting to remove a file descriptor that was never registered causes a *KeyError* exception to be raised.

`poll.`**`poll`**([*timeout*])

Polls the set of registered file descriptors, and returns a possibly-empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with bits set for the reported events for that descriptor — `POLLIN` for waiting input, `POLLOUT` to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, negative, or *None*, the call will block until there is an event for this poll object.

Alterado na versão 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

19.4.4 Kqueue Objects

kqueue.**close**()
Close the control file descriptor of the kqueue object.

kqueue.**closed**
`True` if the kqueue object is closed.

kqueue.**fileno**()
Return the file descriptor number of the control fd.

kqueue.**fromfd**(*fd*)
Create a kqueue object from a given file descriptor.

kqueue.**control**(*changelist*, *max_events*[, *timeout*]) → eventlist
Low level interface to kevent

- changelist must be an iterable of kevent objects or `None`
- max_events must be 0 or a positive integer
- timeout in seconds (floats possible); the default is `None`, to wait forever

Alterado na versão 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising *InterruptedError*.

19.4.5 Kevent Objects

https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

kevent.**ident**
Value used to identify the event. The interpretation depends on the filter but it's usually the file descriptor. In the constructor ident can either be an int or an object with a *fileno()* method. kevent stores the integer internally.

kevent.**filter**
Name of the kernel filter.

Constante	Significado
KQ_FILTER_READ	Takes a descriptor and returns whenever there is data available to read
KQ_FILTER_WRITE	Takes a descriptor and returns whenever there is data available to write
KQ_FILTER_AIO	AIO requests
KQ_FILTER_VNODE	Returns when one or more of the requested events watched in *fflag* occurs
KQ_FILTER_PROC	Watch for events on a process id
KQ_FILTER_NETDEV	Watch for events on a network device [not available on Mac OS X]
KQ_FILTER_SIGNAL	Returns whenever the watched signal is delivered to the process
KQ_FILTER_TIMER	Establishes an arbitrary timer

kevent.**flags**
Filter action.

Constante	Significado
KQ_EV_ADD	Adds or modifies an event
KQ_EV_DELETE	Removes an event from the queue
KQ_EV_ENABLE	Permitscontrol() to returns the event
KQ_EV_DISABLE	Disablesevent
KQ_EV_ONESHOT	Removes event after first occurrence
KQ_EV_CLEAR	Reset the state after an event is retrieved
KQ_EV_SYSFLAGS	internal event
KQ_EV_FLAG1	internal event
KQ_EV_EOF	Filter specific EOF condition
KQ_EV_ERROR	See return values

kevent.**fflags**
Filter specific flags.

KQ_FILTER_READ and KQ_FILTER_WRITE filter flags:

Constante	Significado
KQ_NOTE_LOWAT	low water mark of a socket buffer

KQ_FILTER_VNODE filter flags:

Constante	Significado
KQ_NOTE_DELETE	*unlink()* was called
KQ_NOTE_WRITE	a write occurred
KQ_NOTE_EXTEND	the file was extended
KQ_NOTE_ATTRIB	an attribute was changed
KQ_NOTE_LINK	the link count has changed
KQ_NOTE_RENAME	the file was renamed
KQ_NOTE_REVOKE	access to the file was revoked

KQ_FILTER_PROC filter flags:

Constante	Significado
KQ_NOTE_EXIT	the process has exited
KQ_NOTE_FORK	the process has called *fork()*
KQ_NOTE_EXEC	the process has executed a new process
KQ_NOTE_PCTRLMASK	internal filter flag
KQ_NOTE_PDATAMASK	internal filter flag
KQ_NOTE_TRACK	follow a process across *fork()*
KQ_NOTE_CHILD	returned on the child process for *NOTE_TRACK*
KQ_NOTE_TRACKERR	unable to attach to a child

KQ_FILTER_NETDEV filter flags (not available on Mac OS X):

Constante	Significado
KQ_NOTE_LINKUP	link is up
KQ_NOTE_LINKDOWN	link is down
KQ_NOTE_LINKINV	estado do link é inválido

`kevent.`**`data`**
Filter specific data.

`kevent.`**`udata`**
User defined value.

19.5 `selectors` — High-level I/O multiplexing

Novo na versão 3.4.

Código Fonte: Lib/selectors.py

19.5.1 Introdução

This module allows high-level and efficient I/O multiplexing, built upon the *`select`* module primitives. Users are encouraged to use this module instead, unless they want precise control over the OS-level primitives used.

It defines a *`BaseSelector`* abstract base class, along with several concrete implementations (*`KqueueSelector`*, *`EpollSelector`*...), that can be used to wait for I/O readiness notification on multiple file objects. In the following, "file object" refers to any object with a `fileno()` method, or a raw file descriptor. See *file object*.

`DefaultSelector` is an alias to the most efficient implementation available on the current platform: this should be the default choice for most users.

Nota: The type of file objects supported depends on the platform: on Windows, sockets are supported, but not pipes, whereas on Unix, both are supported (some other types may be supported as well, such as fifos or special file devices).

Ver também:

`select` Low-level I/O multiplexing module.

19.5.2 Classes

Classes hierarchy:

```
BaseSelector
+-- SelectSelector
+-- PollSelector
+-- EpollSelector
+-- DevpollSelector
+-- KqueueSelector
```

In the following, *events* is a bitwise mask indicating which I/O events should be waited for on a given file object. It can be a combination of the modules constants below:

Constante	Significado
`EVENT_READ`	Disponível para leitura
`EVENT_WRITE`	Disponível para escrita

class selectors.**SelectorKey**

A *SelectorKey* is a *namedtuple* used to associate a file object to its underlying file descriptor, selected event mask and attached data. It is returned by several *BaseSelector* methods.

fileobj

File object registered.

fd

O descritor de arquivo subjacente.

events

Events that must be waited for on this file object.

data

Optional opaque data associated to this file object: for example, this could be used to store a per-client session ID.

class selectors.**BaseSelector**

A *BaseSelector* is used to wait for I/O event readiness on multiple file objects. It supports file stream registration, unregistration, and a method to wait for I/O events on those streams, with an optional timeout. It's an abstract base class, so cannot be instantiated. Use *DefaultSelector* instead, or one of *SelectSelector*, *KqueueSelector* etc. if you want to specifically use an implementation, and your platform supports it. *BaseSelector* and its concrete implementations support the *context manager* protocol.

abstractmethod register(*fileobj*, *events*, *data=None*)

Registra um objeto arquivo para seleção, monitorando-o para eventos de I/O

fileobj is the file object to monitor. It may either be an integer file descriptor or an object with a `fileno()` method. *events* is a bitwise mask of events to monitor. *data* is an opaque object.

This returns a new *SelectorKey* instance, or raises a *ValueError* in case of invalid event mask or file descriptor, or *KeyError* if the file object is already registered.

abstractmethod unregister(*fileobj*)

Unregister a file object from selection, removing it from monitoring. A file object shall be unregistered prior to being closed.

fileobj must be a file object previously registered.

This returns the associated *SelectorKey* instance, or raises a *KeyError* if *fileobj* is not registered. It will raise *ValueError* if *fileobj* is invalid (e.g. it has no `fileno()` method or its `fileno()` method has an invalid return value).

modify(*fileobj*, *events*, *data=None*)

Change a registered file object's monitored events or attached data.

This is equivalent to `BaseSelector.unregister(fileobj)()` followed by `BaseSelector.register(fileobj, events, data)()`, except that it can be implemented more efficiently.

This returns a new *SelectorKey* instance, or raises a *ValueError* in case of invalid event mask or file descriptor, or *KeyError* if the file object is not registered.

abstractmethod select(*timeout=None*)

Wait until some registered file objects become ready, or the timeout expires.

If `timeout > 0`, this specifies the maximum wait time, in seconds. If `timeout <= 0`, the call won't block, and will report the currently ready file objects. If *timeout* is `None`, the call will block until a monitored file object becomes ready.

This returns a list of `(key, events)` tuples, one for each ready file object.

key is the *SelectorKey* instance corresponding to a ready file object. *events* is a bitmask of events ready on this file object.

Nota: This method can return before any file object becomes ready or the timeout has elapsed if the current process receives a signal: in this case, an empty list will be returned.

Alterado na versão 3.5: The selector is now retried with a recomputed timeout when interrupted by a signal if the signal handler did not raise an exception (see **PEP 475** for the rationale), instead of returning an empty list of events before the timeout.

close()
Close the selector.

This must be called to make sure that any underlying resource is freed. The selector shall not be used once it has been closed.

get_key(*fileobj*)
Return the key associated with a registered file object.

This returns the *SelectorKey* instance associated to this file object, or raises *KeyError* if the file object is not registered.

abstractmethod get_map()
Return a mapping of file objects to selector keys.

This returns a *Mapping* instance mapping registered file objects to their associated *SelectorKey* instance.

class selectors.**DefaultSelector**
The default selector class, using the most efficient implementation available on the current platform. This should be the default choice for most users.

class selectors.**SelectSelector**
select.select()-based selector.

class selectors.**PollSelector**
select.poll()-based selector.

class selectors.**EpollSelector**
select.epoll()-based selector.

fileno()
This returns the file descriptor used by the underlying *select.epoll()* object.

class selectors.**DevpollSelector**
select.devpoll()-based selector.

fileno()
This returns the file descriptor used by the underlying *select.devpoll()* object.

Novo na versão 3.5.

class selectors.**KqueueSelector**
select.kqueue()-based selector.

fileno()
This returns the file descriptor used by the underlying *select.kqueue()* object.

19.5.3 Exemplos

Here is a simple echo server implementation:

```
import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()  # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

def read(conn, mask):
    data = conn.recv(1000)  # Should be ready
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # Hope it won't block
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

19.6 `asyncore` — Asynchronous socket handler

Código Fonte: Lib/asyncore.py

Obsoleto desde a versão 3.6: Please use *`asyncio`* instead.

Nota: This module exists for backwards compatibility only. For new code we recommend using *`asyncio`*.

This module provides the basic infrastructure for writing asynchronous socket service clients and servers.

There are only two ways to have a program on a single processor do "more than one thing at a time." Multi-threaded programming is the simplest and most popular way to do it, but there is another very different technique, that lets you have nearly all the advantages of multi-threading, without actually using multiple threads. It's really only practical if your program is largely I/O bound. If your program is processor bound, then pre-emptive scheduled threads are probably what you really need. Network servers are rarely processor bound, however.

If your operating system supports the `select()` system call in its I/O library (and nearly all do), then you can use it to juggle multiple communication channels at once; doing other work while your I/O is taking place in the "background." Although this strategy can seem strange and complex, especially at first, it is in many ways easier to understand and control than multi-threaded programming. The *asyncore* module solves many of the difficult problems for you, making the task of building sophisticated high-performance network servers and clients a snap. For "conversational" applications and protocols the companion *asynchat* module is invaluable.

The basic idea behind both modules is to create one or more network *channels*, instances of class *asyncore.dispatcher* and *asynchat.async_chat*. Creating the channels adds them to a global map, used by the *loop()* function if you do not provide it with your own *map*.

Once the initial channel(s) is(are) created, calling the *loop()* function activates channel service, which continues until the last channel (including any that have been added to the map during asynchronous service) is closed.

`asyncore.`**`loop`**`(`[*timeout*[, *use_poll*[, *map*[, *count*]]]]`)`

Enter a polling loop that terminates after count passes or all open channels have been closed. All arguments are optional. The *count* parameter defaults to `None`, resulting in the loop terminating only when all channels have been closed. The *timeout* argument sets the timeout parameter for the appropriate *select()* or *poll()* call, measured in seconds; the default is 30 seconds. The *use_poll* parameter, if true, indicates that *poll()* should be used in preference to *select()* (the default is `False`).

The *map* parameter is a dictionary whose items are the channels to watch. As channels are closed they are deleted from their map. If *map* is omitted, a global map is used. Channels (instances of *asyncore.dispatcher*, *asynchat.async_chat* and subclasses thereof) can freely be mixed in the map.

class `asyncore.`**`dispatcher`**

The *dispatcher* class is a thin wrapper around a low-level socket object. To make it more useful, it has a few methods for event-handling which are called from the asynchronous loop. Otherwise, it can be treated as a normal non-blocking socket object.

The firing of low-level events at certain times or in certain connection states tells the asynchronous loop that certain higher-level events have taken place. For example, if we have asked for a socket to connect to another host, we know that the connection has been made when the socket becomes writable for the first time (at this point you know that you may write to it with the expectation of success). The implied higher-level events are:

Evento	Description (descrição)
`handle_connect()`	Implied by the first read or write event
`handle_close()`	Implied by a read event with no data available
`handle_accepted()`	Implied by a read event on a listening socket

During asynchronous processing, each mapped channel's *readable()* and *writable()* methods are used to determine whether the channel's socket should be added to the list of channels `select()`ed or `poll()`ed for read and write events.

Thus, the set of channel events is larger than the basic socket events. The full set of methods that can be overridden in your subclass follows:

`handle_read``()`

Called when the asynchronous loop detects that a `read()` call on the channel's socket will succeed.

`handle_write``()`

Called when the asynchronous loop detects that a writable socket can be written. Often this method will implement the necessary buffering for performance. For example:

```
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

handle_expt()
Chamado quando há dados fora da banda (00B) para uma conexão socket. Isso quase nunca acontece, como a 00B é suportada com tenacidade e raramente usada.

handle_connect()
Called when the active opener's socket actually makes a connection. Might send a "welcome" banner, or initiate a protocol negotiation with the remote endpoint, for example.

handle_close()
Called when the socket is closed.

handle_error()
Called when an exception is raised and not otherwise handled. The default version prints a condensed traceback.

handle_accept()
Called on listening channels (passive openers) when a connection can be established with a new remote endpoint that has issued a *connect()* call for the local endpoint. Deprecated in version 3.2; use *handle_accepted()* instead.

Obsoleto desde a versão 3.2.

handle_accepted(*sock*, *addr*)
Called on listening channels (passive openers) when a connection has been established with a new remote endpoint that has issued a *connect()* call for the local endpoint. *sock* is a *new* socket object usable to send and receive data on the connection, and *addr* is the address bound to the socket on the other end of the connection.

Novo na versão 3.2.

readable()
Called each time around the asynchronous loop to determine whether a channel's socket should be added to the list on which read events can occur. The default method simply returns `True`, indicating that by default, all channels will be interested in read events.

writable()
Called each time around the asynchronous loop to determine whether a channel's socket should be added to the list on which write events can occur. The default method simply returns `True`, indicating that by default, all channels will be interested in write events.

In addition, each channel delegates or extends many of the socket methods. Most of these are nearly identical to their socket partners.

create_socket(*family=socket.AF_INET*, *type=socket.SOCK_STREAM*)
This is identical to the creation of a normal socket, and will use the same options for creation. Refer to the *socket* documentation for information on creating sockets.

Alterado na versão 3.3: *family* and *type* arguments can be omitted.

connect(*address*)
As with the normal socket object, *address* is a tuple with the first element the host to connect to, and the second the port number.

send(*data*)
Send *data* to the remote end-point of the socket.

recv(*buffer_size*)
Read at most *buffer_size* bytes from the socket's remote end-point. An empty bytes object implies that the channel has been closed from the other end.

Note that *recv()* may raise *BlockingIOError* , even though *select.select()* or *select.poll()* has reported the socket ready for reading.

listen(*backlog*)
Listen for connections made to the socket. The *backlog* argument specifies the maximum number of queued connections and should be at least 1; the maximum value is system-dependent (usually 5).

bind(*address*)
Bind the socket to *address*. The socket must not already be bound. (The format of *address* depends on the address family — refer to the *socket* documentation for more information.) To mark the socket as re-usable (setting the `SO_REUSEADDR` option), call the *dispatcher* object's `set_reuse_addr()` method.

accept()
Accept a connection. The socket must be bound to an address and listening for connections. The return value can be either `None` or a pair `(conn, address)` where *conn* is a *new* socket object usable to send and receive data on the connection, and *address* is the address bound to the socket on the other end of the connection. When `None` is returned it means the connection didn't take place, in which case the server should just ignore this event and keep listening for further incoming connections.

close()
Close the socket. All future operations on the socket object will fail. The remote end-point will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected.

class asyncore.**dispatcher_with_send**
A *dispatcher* subclass which adds simple buffered output capability, useful for simple clients. For more sophisticated usage use *asynchat.async_chat*.

class asyncore.**file_dispatcher**
A file_dispatcher takes a file descriptor or *file object* along with an optional map argument and wraps it for use with the `poll()` or `loop()` functions. If provided a file object or anything with a `fileno()` method, that method will be called and passed to the *file_wrapper* constructor.

Disponibilidade: Unix.

class asyncore.**file_wrapper**
A file_wrapper takes an integer file descriptor and calls *os.dup()* to duplicate the handle so that the original handle may be closed independently of the file_wrapper. This class implements sufficient methods to emulate a socket for use by the *file_dispatcher* class.

Disponibilidade: Unix.

19.6.1 asyncore Example basic HTTP client

Here is a very basic HTTP client that uses the *dispatcher* class to implement its socket handling:

```
import asyncore

class HTTPClient(asyncore.dispatcher):

    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.connect( (host, 80) )
        self.buffer = bytes('GET %s HTTP/1.0\r\nHost: %s\r\n\r\n' %
                            (path, host), 'ascii')

    def handle_connect(self):
        pass

    def handle_close(self):
        self.close()
```

(continua na próxima página)

(continuação da página anterior)

```
    def handle_read(self):
        print(self.recv(8192))

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]


client = HTTPClient('www.python.org', '/')
asyncore.loop()
```

19.6.2 asyncore Example basic echo server

Here is a basic echo server that uses the *dispatcher* class to accept connections and dispatches the incoming connections to a handler:

```
import asyncore

class EchoHandler(asyncore.dispatcher_with_send):

    def handle_read(self):
        data = self.recv(8192)
        if data:
            self.send(data)

class EchoServer(asyncore.dispatcher):

    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.set_reuse_addr()
        self.bind((host, port))
        self.listen(5)

    def handle_accepted(self, sock, addr):
        print('Incoming connection from %s' % repr(addr))
        handler = EchoHandler(sock)

server = EchoServer('localhost', 8080)
asyncore.loop()
```

19.7 asynchat — Asynchronous socket command/response handler

Código Fonte: Lib/asynchat.py

Obsoleto desde a versão 3.6: Please use *asyncio* instead.

Nota: This module exists for backwards compatibility only. For new code we recommend using *asyncio*.

This module builds on the *asyncore* infrastructure, simplifying asynchronous clients and servers and making it easier to handle protocols whose elements are terminated by arbitrary strings, or are of variable length. *asynchat* defines the abstract class *async_chat* that you subclass, providing implementations of the `collect_incoming_data()` and `found_terminator()` methods. It uses the same asynchronous loop as *asyncore*, and the two types of channel, *asyncore.dispatcher* and *asynchat.async_chat*, can freely be mixed in the channel map. Typically an *asyncore.dispatcher* server channel generates new *asynchat.async_chat* channel objects as it receives incoming connection requests.

class asynchat.**async_chat**

This class is an abstract subclass of *asyncore.dispatcher*. To make practical use of the code you must subclass *async_chat*, providing meaningful *collect_incoming_data()* and *found_terminator()* methods. The *asyncore.dispatcher* methods can be used, although not all make sense in a message/response context.

Like *asyncore.dispatcher*, *async_chat* defines a set of events that are generated by an analysis of socket conditions after a `select()` call. Once the polling loop has been started the *async_chat* object's methods are called by the event-processing framework with no action on the part of the programmer.

Two class attributes can be modified, to improve performance, or possibly even to conserve memory.

ac_in_buffer_size

The asynchronous input buffer size (default `4096`).

ac_out_buffer_size

The asynchronous output buffer size (default `4096`).

Unlike *asyncore.dispatcher*, *async_chat* allows you to define a FIFO queue of *producers*. A producer need have only one method, `more()`, which should return data to be transmitted on the channel. The producer indicates exhaustion (*i.e.* that it contains no more data) by having its `more()` method return the empty bytes object. At this point the *async_chat* object removes the producer from the queue and starts using the next producer, if any. When the producer queue is empty the `handle_write()` method does nothing. You use the channel object's *set_terminator()* method to describe how to recognize the end of, or an important breakpoint in, an incoming transmission from the remote endpoint.

To build a functioning *async_chat* subclass your input methods *collect_incoming_data()* and *found_terminator()* must handle the data that the channel receives asynchronously. The methods are described below.

async_chat.**close_when_done**()

Pushes a `None` on to the producer queue. When this producer is popped off the queue it causes the channel to be closed.

async_chat.**collect_incoming_data**(*data*)

Called with *data* holding an arbitrary amount of received data. The default method, which must be overridden, raises a *NotImplementedError* exception.

async_chat.**discard_buffers**()

In emergencies this method will discard any data held in the input and/or output buffers and the producer queue.

async_chat.**found_terminator**()
Called when the incoming data stream matches the termination condition set by *set_terminator()*. The default method, which must be overridden, raises a *NotImplementedError* exception. The buffered input data should be available via an instance attribute.

async_chat.**get_terminator**()
Returns the current terminator for the channel.

async_chat.**push**(*data*)
Pushes data on to the channel's queue to ensure its transmission. This is all you need to do to have the channel write the data out to the network, although it is possible to use your own producers in more complex schemes to implement encryption and chunking, for example.

async_chat.**push_with_producer**(*producer*)
Takes a producer object and adds it to the producer queue associated with the channel. When all currently-pushed producers have been exhausted the channel will consume this producer's data by calling its `more()` method and send the data to the remote endpoint.

async_chat.**set_terminator**(*term*)
Sets the terminating condition to be recognized on the channel. `term` may be any of three types of value, corresponding to three different ways to handle incoming protocol data.

term	Description (descrição)
string	Will call *found_terminator()* when the string is found in the input stream
integer	Will call *found_terminator()* when the indicated number of characters have been received
`None`	The channel continues to collect data forever

Note that any data following the terminator will be available for reading by the channel after *found_terminator()* is called.

19.7.1 Exemplo com asynchat

The following partial example shows how HTTP requests can be read with *async_chat*. A web server might create an `http_request_handler` object for each incoming client connection. Notice that initially the channel terminator is set to match the blank line at the end of the HTTP headers, and a flag indicates that the headers are being read.

Once the headers have been read, if the request is of type POST (indicating that further data are present in the input stream) then the `Content-Length:` header is used to set a numeric terminator to read the right amount of data from the channel.

The `handle_request()` method is called once all relevant input has been marshalled, after setting the channel terminator to `None` to ensure that any extraneous data sent by the web client are ignored.

```
import asynchat

class http_request_handler(asynchat.async_chat):

    def __init__(self, sock, addr, sessions, log):
        asynchat.async_chat.__init__(self, sock=sock)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = b""
        self.set_terminator(b"\r\n\r\n")
        self.reading_headers = True
        self.handling = False
```

(continua na próxima página)

(continuação da página anterior)

```
        self.cgi_data = None
        self.log = log

    def collect_incoming_data(self, data):
        """Buffer the data"""
        self.ibuffer.append(data)

    def found_terminator(self):
        if self.reading_headers:
            self.reading_headers = False
            self.parse_headers(b"".join(self.ibuffer))
            self.ibuffer = []
            if self.op.upper() == b"POST":
                clen = self.headers.getheader("content-length")
                self.set_terminator(int(clen))
            else:
                self.handling = True
                self.set_terminator(None)
                self.handle_request()
        elif not self.handling:
            self.set_terminator(None)  # browsers sometimes over-send
            self.cgi_data = parse(self.headers, b"".join(self.ibuffer))
            self.handling = True
            self.ibuffer = []
            self.handle_request()
```

19.8 `signal` — Set handlers for asynchronous events

This module provides mechanisms to use signal handlers in Python.

19.8.1 Regras gerais

The *signal.signal()* function allows defining custom handlers to be executed when a signal is received. A small number of default handlers are installed: *SIGPIPE* is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions) and *SIGINT* is translated into a *KeyboardInterrupt* exception if the parent process has not changed it.

A handler for a particular signal, once set, remains installed until it is explicitly reset (Python emulates the BSD style interface regardless of the underlying implementation), with the exception of the handler for *SIGCHLD*, which follows the underlying implementation.

Execution of Python signal handlers

A Python signal handler does not get executed inside the low-level (C) signal handler. Instead, the low-level signal handler sets a flag which tells the *virtual machine* to execute the corresponding Python signal handler at a later point(for example at the next *bytecode* instruction). This has consequences:

- It makes little sense to catch synchronous errors like `SIGFPE` or `SIGSEGV` that are caused by an invalid operation in C code. Python will return from the signal handler to the C code, which is likely to raise the same signal again, causing Python to apparently hang. From Python 3.3 onwards, you can use the `faulthandler` module to report on synchronous errors.
- A long-running calculation implemented purely in C (such as regular expression matching on a large body of text) may run uninterrupted for an arbitrary amount of time, regardless of any signals received. The Python signal handlers will be called when the calculation finishes.

Signals and threads

Python signal handlers are always executed in the main Python thread, even if the signal was received in another thread. This means that signals can't be used as a means of inter-thread communication. You can use the synchronization primitives from the `threading` module instead.

Besides, only the main thread is allowed to set a new signal handler.

19.8.2 Conteúdo do módulo

Alterado na versão 3.5: signal (SIG*), handler (`SIG_DFL`, `SIG_IGN`) and sigmask (`SIG_BLOCK`, `SIG_UNBLOCK`, `SIG_SETMASK`) related constants listed below were turned into `enums`. `getsignal()`, `pthread_sigmask()`, `sigpending()` and `sigwait()` functions return human-readable `enums`.

The variables defined in the `signal` module are:

`signal.`**`SIG_DFL`**
: This is one of two standard signal handling options; it will simply perform the default function for the signal. For example, on most systems the default action for `SIGQUIT` is to dump core and exit, while the default action for `SIGCHLD` is to simply ignore it.

`signal.`**`SIG_IGN`**
: This is another standard signal handler, which will simply ignore the given signal.

`signal.`**`SIGABRT`**
: Abort signal from `abort(3)`.

`signal.`**`SIGALRM`**
: Timer signal from `alarm(2)`.

  Disponibilidade: Unix.

`signal.`**`SIGBREAK`**
: Interrupt from keyboard (CTRL + BREAK).

  Disponibilidade: Windows.

`signal.`**`SIGBUS`**
: Bus error (bad memory access).

  Disponibilidade: Unix.

signal.**SIGCHLD**
Child process stopped or terminated.

Disponibilidade: Windows.

signal.**SIGCLD**
Alias to *SIGCHLD*.

signal.**SIGCONT**
Continue the process if it is currently stopped

Disponibilidade: Unix.

signal.**SIGFPE**
Floating-point exception. For example, division by zero.

Ver também:

ZeroDivisionError is raised when the second argument of a division or modulo operation is zero.

signal.**SIGHUP**
Hangup detected on controlling terminal or death of controlling process.

Disponibilidade: Unix.

signal.**SIGILL**
Instrução ilegal.

signal.**SIGINT**
Interrupt from keyboard (CTRL + C).

Default action is to raise *KeyboardInterrupt*.

signal.**SIGKILL**
Kill signal.

It cannot be caught, blocked, or ignored.

Disponibilidade: Unix.

signal.**SIGPIPE**
Broken pipe: write to pipe with no readers.

Default action is to ignore the signal.

Disponibilidade: Unix.

signal.**SIGSEGV**
Segmentation fault: invalid memory reference.

signal.**SIGTERM**
Termination signal.

signal.**SIGUSR1**
User-defined signal 1.

Disponibilidade: Unix.

signal.**SIGUSR2**
User-defined signal 2.

Disponibilidade: Unix.

signal.**SIGWINCH**
Window resize signal.

Disponibilidade: Unix.

SIG*
All the signal numbers are defined symbolically. For example, the hangup signal is defined as *signal.SIGHUP*; the variable names are identical to the names used in C programs, as found in <signal.h>. The Unix man page for 'signal()' lists the existing signals (on some systems this is *signal(2)*, on others the list is in *signal(7)*). Note that not all systems define the same set of signal names; only those names defined by the system are defined by this module.

signal.**CTRL_C_EVENT**
The signal corresponding to the Ctrl+C keystroke event. This signal can only be used with *os.kill()*.

Disponibilidade: Windows.

Novo na versão 3.2.

signal.**CTRL_BREAK_EVENT**
The signal corresponding to the Ctrl+Break keystroke event. This signal can only be used with *os.kill()*.

Disponibilidade: Windows.

Novo na versão 3.2.

signal.**NSIG**
One more than the number of the highest signal number.

signal.**ITIMER_REAL**
Decrements interval timer in real time, and delivers *SIGALRM* upon expiration.

signal.**ITIMER_VIRTUAL**
Decrements interval timer only when the process is executing, and delivers SIGVTALRM upon expiration.

signal.**ITIMER_PROF**
Decrements interval timer both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

signal.**SIG_BLOCK**
A possible value for the *how* parameter to *pthread_sigmask()* indicating that signals are to be blocked.

Novo na versão 3.3.

signal.**SIG_UNBLOCK**
A possible value for the *how* parameter to *pthread_sigmask()* indicating that signals are to be unblocked.

Novo na versão 3.3.

signal.**SIG_SETMASK**
A possible value for the *how* parameter to *pthread_sigmask()* indicating that the signal mask is to be replaced.

Novo na versão 3.3.

The *signal* module defines one exception:

exception signal.**ItimerError**
Raised to signal an error from the underlying *setitimer()* or *getitimer()* implementation. Expect this error if an invalid interval timer or a negative time is passed to *setitimer()*. This error is a subtype of *OSError*.

Novo na versão 3.3: This error used to be a subtype of *IOError*, which is now an alias of *OSError*.

The *signal* module defines the following functions:

signal.**alarm**(*time*)
If *time* is non-zero, this function requests that a *SIGALRM* signal be sent to the process in *time* seconds. Any previously scheduled alarm is canceled (only one alarm can be scheduled at any time). The returned value is then

the number of seconds before any previously set alarm was to have been delivered. If *time* is zero, no alarm is scheduled, and any scheduled alarm is canceled. If the return value is zero, no alarm is currently scheduled.

Availability: Unix. See the man page *alarm(2)* for further information.

signal.**getsignal**(*signalnum*)
Return the current signal handler for the signal *signalnum*. The returned value may be a callable Python object, or one of the special values *signal.SIG_IGN*, *signal.SIG_DFL* or *None*. Here, *signal.SIG_IGN* means that the signal was previously ignored, *signal.SIG_DFL* means that the default way of handling the signal was previously in use, and None means that the previous signal handler was not installed from Python.

signal.**pause**()
Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing.

Availability: Unix. See the man page *signal(2)* for further information.

See also *sigwait()*, *sigwaitinfo()*, *sigtimedwait()* and *sigpending()*.

signal.**pthread_kill**(*thread_id*, *signalnum*)
Send the signal *signalnum* to the thread *thread_id*, another thread in the same process as the caller. The target thread can be executing any code (Python or not). However, if the target thread is executing the Python interpreter, the Python signal handlers will be *executed by the main thread*. Therefore, the only point of sending a signal to a particular Python thread would be to force a running system call to fail with *InterruptedError*.

Use *threading.get_ident()* or the *ident* attribute of *threading.Thread* objects to get a suitable value for *thread_id*.

If *signalnum* is 0, then no signal is sent, but error checking is still performed; this can be used to check if the target thread is still running.

Availability: Unix. See the man page *pthread_kill(3)* for further information.

See also *os.kill()*.

Novo na versão 3.3.

signal.**pthread_sigmask**(*how*, *mask*)
Fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller. Return the old signal mask as a set of signals.

The behavior of the call is dependent on the value of *how*, as follows.

- *SIG_BLOCK*: The set of blocked signals is the union of the current set and the *mask* argument.
- *SIG_UNBLOCK*: The signals in *mask* are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.
- *SIG_SETMASK*: The set of blocked signals is set to the *mask* argument.

mask is a set of signal numbers (e.g. {*signal.SIGINT*, *signal.SIGTERM*}). Use `range(1, signal.NSIG)` for a full mask including all signals.

For example, `signal.pthread_sigmask(signal.SIG_BLOCK, [])` reads the signal mask of the calling thread.

SIGKILL and SIGSTOP cannot be blocked.

Availability: Unix. See the man page *sigprocmask(3)* and *pthread_sigmask(3)* for further information.

Veja também *pause()*, *sigpending()* e *sigwait()*.

Novo na versão 3.3.

signal.**setitimer**(*which*, *seconds*, *interval=0.0*)
Sets given interval timer (one of *signal.ITIMER_REAL*, *signal.ITIMER_VIRTUAL* or *signal.ITIMER_PROF*) specified by *which* to fire after *seconds* (float is accepted, different from *alarm()*) and after that every *interval* seconds (if *interval* is non-zero). The interval timer specified by *which* can be cleared by setting *seconds* to zero.

When an interval timer fires, a signal is sent to the process. The signal sent is dependent on the timer being used; *signal.ITIMER_REAL* will deliver *SIGALRM*, *signal.ITIMER_VIRTUAL* sends SIGVTALRM, and *signal.ITIMER_PROF* will deliver SIGPROF.

The old values are returned as a tuple: (delay, interval).

Attempting to pass an invalid interval timer will cause an *ItimerError*.

Disponibilidade: Unix.

signal.**getitimer**(*which*)
Returns current value of a given interval timer specified by *which*.

Disponibilidade: Unix.

signal.**set_wakeup_fd**(*fd*, *, *warn_on_full_buffer=True*)
Set the wakeup file descriptor to *fd*. When a signal is received, the signal number is written as a single byte into the fd. This can be used by a library to wakeup a poll or select call, allowing the signal to be fully processed.

The old wakeup fd is returned (or -1 if file descriptor wakeup was not enabled). If *fd* is -1, file descriptor wakeup is disabled. If not -1, *fd* must be non-blocking. It is up to the library to remove any bytes from *fd* before calling poll or select again.

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a *ValueError* exception to be raised.

There are two common ways to use this function. In both approaches, you use the fd to wake up when a signal arrives, but then they differ in how they determine *which* signal or signals have arrived.

In the first approach, we read the data out of the fd's buffer, and the byte values give you the signal numbers. This is simple, but in rare cases it can run into a problem: generally the fd will have a limited amount of buffer space, and if too many signals arrive too quickly, then the buffer may become full, and some signals may be lost. If you use this approach, then you should set `warn_on_full_buffer=True`, which will at least cause a warning to be printed to stderr when signals are lost.

In the second approach, we use the wakeup fd *only* for wakeups, and ignore the actual byte values. In this case, all we care about is whether the fd's buffer is empty or non-empty; a full buffer doesn't indicate a problem at all. If you use this approach, then you should set `warn_on_full_buffer=False`, so that your users are not confused by spurious warning messages.

Alterado na versão 3.5: No Windows, a função agora também suporta manipuladores de socket.

Alterado na versão 3.7: Added `warn_on_full_buffer` parameter.

signal.**siginterrupt**(*signalnum*, *flag*)
Change system call restart behaviour: if *flag* is *False*, system calls will be restarted when interrupted by signal *signalnum*, otherwise system calls will be interrupted. Returns nothing.

Availability: Unix. See the man page *siginterrupt(3)* for further information.

Note that installing a signal handler with *signal()* will reset the restart behaviour to interruptible by implicitly calling `siginterrupt()` with a true *flag* value for the given signal.

signal.**signal**(*signalnum*, *handler*)
Set the handler for signal *signalnum* to the function *handler*. *handler* can be a callable Python object taking two arguments (see below), or one of the special values *signal.SIG_IGN* or *signal.SIG_DFL*. The previous sig-

nal handler will be returned (see the description of *getsignal()* above). (See the Unix man page *signal(2)* for further information.)

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a *ValueError* exception to be raised.

The *handler* is called with two arguments: the signal number and the current stack frame (`None` or a frame object; for a description of frame objects, see the description in the type hierarchy or see the attribute descriptions in the *inspect* module).

On Windows, *signal()* can only be called with *SIGABRT*, *SIGFPE*, *SIGILL*, *SIGINT*, *SIGSEGV*, *SIGTERM*, or *SIGBREAK*. A *ValueError* will be raised in any other case. Note that not all systems define the same set of signal names; an *AttributeError* will be raised if a signal name is not defined as `SIG*` module level constant.

`signal.`**`sigpending`**`()`
Examine the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked). Return the set of the pending signals.

Availability: Unix. See the man page *sigpending(2)* for further information.

Veja também *pause()*, *sigpending()* e *sigwait()*.

Novo na versão 3.3.

`signal.`**`sigwait`**`(`*sigset*`)`
Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal (removes it from the pending list of signals), and returns the signal number.

Availability: Unix. See the man page *sigwait(3)* for further information.

See also *pause()*, *pthread_sigmask()*, *sigpending()*, *sigwaitinfo()* and *sigtimedwait()*.

Novo na versão 3.3.

`signal.`**`sigwaitinfo`**`(`*sigset*`)`
Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal and removes it from the pending list of signals. If one of the signals in *sigset* is already pending for the calling thread, the function will return immediately with information about that signal. The signal handler is not called for the delivered signal. The function raises an *InterruptedError* if it is interrupted by a signal that is not in *sigset*.

The return value is an object representing the data contained in the `siginfo_t` structure, namely: `si_signo`, `si_code`, `si_errno`, `si_pid`, `si_uid`, `si_status`, `si_band`.

Availability: Unix. See the man page *sigwaitinfo(2)* for further information.

See also *pause()*, *sigwait()* and *sigtimedwait()*.

Novo na versão 3.3.

Alterado na versão 3.5: The function is now retried if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see **PEP 475** for the rationale).

`signal.`**`sigtimedwait`**`(`*sigset*, *timeout*`)`
Like *sigwaitinfo()*, but takes an additional *timeout* argument specifying a timeout. If *timeout* is specified as `0`, a poll is performed. Returns *None* if a timeout occurs.

Availability: Unix. See the man page *sigtimedwait(2)* for further information.

See also *pause()*, *sigwait()* and *sigwaitinfo()*.

Novo na versão 3.3.

Alterado na versão 3.5: The function is now retried with the recomputed *timeout* if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see **PEP 475** for the rationale).

19.8.3 Exemplo

Here is a minimal example program. It uses the *alarm()* function to limit the time spent waiting to open a file; this is useful if the file is for a serial device that may not be turned on, which would normally cause the *os.open()* to hang indefinitely. The solution is to set a 5-second alarm before opening the file; if the operation takes too long, the alarm signal will be sent, and the handler raises an exception.

```
import signal, os

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    raise OSError("Couldn't open device!")

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # Disable the alarm
```

19.8.4 Note on SIGPIPE

Piping output of your program to tools like *head(1)* will cause a *SIGPIPE* signal to be sent to your process when the receiver of its standard output closes early. This results in an exception like `BrokenPipeError: [Errno 32] Broken pipe`. To handle this case, wrap your entry point to catch this exception as follows:

```
import os
import sys

def main():
    try:
        # simulate large output (your code replaces this loop)
        for x in range(10000):
            print("y")
        # flush output here to force SIGPIPE to be triggered
        # while inside this try block.
        sys.stdout.flush()
    except BrokenPipeError:
        # Python flushes standard streams on exit; redirect remaining output
        # to devnull to avoid another BrokenPipeError at shutdown
        devnull = os.open(os.devnull, os.O_WRONLY)
        os.dup2(devnull, sys.stdout.fileno())
        sys.exit(1)  # Python exits with error code 1 on EPIPE

if __name__ == '__main__':
    main()
```

Do not set *SIGPIPE*'s disposition to *SIG_DFL* in order to avoid *BrokenPipeError*. Doing that would cause your program to exit unexpectedly also whenever any socket connection is interrupted while your program is still writing to it.

19.9 mmap — Suporte a arquivos com memória mapeada

Objetos de arquivo mapeados na memória se comportam como *bytearray* e como *objetos de arquivo*. Você pode usar objetos mmap na maioria dos lugares onde *bytearray* é esperado; por exemplo, você pode usar o módulo *re* para pesquisar um arquivo mapeado na memória. Você também pode alterar um único byte executando `obj[index] = 97` ou alterar uma subsequência atribuindo a uma fatia: `obj[i1:i2] = b'...'`. Você também pode ler e gravar dados começando na posição atual do arquivo e `seek()` através do arquivo para diferentes posições.

Um arquivo mapeado na memória é criado pelo construtor *mmap*, que é diferente no Unix e no Windows. Nos dois casos, você deve fornecer um descritor de arquivo para um arquivo aberto para atualização. Se você deseja mapear um objeto de arquivo Python existente, use o método `fileno()` para obter o valor correto para o parâmetro *fileno*. Caso contrário, você pode abrir o arquivo usando a função *os.open()*, que retorna um descritor de arquivo diretamente (o arquivo ainda precisa ser fechado quando terminar).

Nota: Se você deseja criar um mapeamento de memória para um arquivo gravável e armazenado em buffer, deve usar *flush()* no arquivo primeiro. Isso é necessário para garantir que as modificações locais nos buffers estejam realmente disponíveis para o mapeamento.

Para as versões Unix e Windows do construtor, *access* pode ser especificado como um parâmetro opcional de palavra-chave. *access* aceita um dos quatro valores: `ACCESS_READ`, `ACCESS_WRITE` ou `ACCESS_COPY` para especificar memória somente leitura, gravação ou cópia na gravação, respectivamente `ACCESS_DEFAULT` para adiar para *prot*. *access* pode ser usado no Unix e no Windows. Se *access* não for especificado, o mmap do Windows retornará um mapeamento de gravação. Os valores iniciais da memória para todos os três tipos de acesso são obtidos do arquivo especificado. A atribuição a um mapa de memória `ACCESS_READ` gera uma exceção *TypeError*. A atribuição a um mapa de memória `ACCESS_WRITE` afeta a memória e o arquivo subjacente. A atribuição a um mapa de memória `ACCESS_COPY` afeta a memória, mas não atualiza o arquivo subjacente.

Alterado na versão 3.7: Adicionada a constante `ACCESS_DEFAULT`.

Para mapear a memória anônima, -1 deve ser passado como o fileno junto com o comprimento.

class mmap.**mmap**(*fileno*, *length*, *tagname=None*, *access=ACCESS_DEFAULT*[, *offset*])

(Versão Windows) Mapeia *length* bytes do arquivo especificado pelo identificador de arquivo *fileno* e cria um objeto mmap. Se *length* for maior que o tamanho atual do arquivo, o arquivo será estendido para conter *length* bytes. Se *length* for `0`, o tamanho máximo do mapa será o tamanho atual do arquivo, exceto que, se o arquivo estiver vazio, o Windows gerará uma exceção (você não poderá criar um mapeamento vazio no Windows).

tagname, se especificado e não `None`, é uma string que fornece um nome de tag para o mapeamento. O Windows permite que você tenha muitos mapeamentos diferentes no mesmo arquivo. Se você especificar o nome de uma marca existente, essa marca será aberta; caso contrário, uma nova marca com esse nome será criada. Se este parâmetro for omitido ou "None", o mapeamento será criado sem um nome. Evitar o uso do parâmetro tag ajudará a manter seu código portátil entre o Unix e o Windows.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of the `ALLOCATIONGRANULARITY`.

class mmap.**mmap**(*fileno*, *length*, *flags=MAP_SHARED*, *prot=PROT_WRITE|PROT_READ*, *access=ACCESS_DEFAULT*[, *offset*])

(Unix version) Maps *length* bytes from the file specified by the file descriptor *fileno*, and returns a mmap object. If *length* is `0`, the maximum length of the map will be the current size of the file when *mmap* is called.

flags specifies the nature of the mapping. `MAP_PRIVATE` creates a private copy-on-write mapping, so changes to the contents of the mmap object will be private to this process, and `MAP_SHARED` creates a mapping that's shared with all other processes mapping the same areas of the file. The default value is `MAP_SHARED`.

prot, if specified, gives the desired memory protection; the two most useful values are `PROT_READ` and `PROT_WRITE`, to specify that the pages may be read or written. *prot* defaults to `PROT_READ | PROT_WRITE`.

access may be specified in lieu of *flags* and *prot* as an optional keyword parameter. It is an error to specify both *flags*, *prot* and *access*. See the description of *access* above for information on how to use this parameter.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of `ALLOCATIONGRANULARITY` which is equal to `PAGESIZE` on Unix systems.

To ensure validity of the created memory mapping the file specified by the descriptor *fileno* is internally automatically synchronized with physical backing store on Mac OS X and OpenVMS.

This example shows a simple way of using *mmap*:

```
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print(mm.readline())  # prints b"Hello Python!\n"
    # read content via slice notation
    print(mm[:5])  # prints b"Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = b" world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print(mm.readline())  # prints b"Hello  world!\n"
    # close the map
    mm.close()
```

mmap can also be used as a context manager in a `with` statement:

```
import mmap

with mmap.mmap(-1, 13) as mm:
    mm.write(b"Hello world!")
```

Novo na versão 3.2: Suporte a gerenciador de contexto.

The next example demonstrates how to create an anonymous map and exchange data between the parent and child processes:

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")

pid = os.fork()

if pid == 0:  # In a child process
    mm.seek(0)
```

(continua na próxima página)

(continuação da página anterior)

```
    print(mm.readline())

    mm.close()
```

Memory-mapped file objects support the following methods:

close()
Closes the mmap. Subsequent calls to other methods of the object will result in a ValueError exception being raised. This will not close the open file.

closed
`True` if the file is closed.

Novo na versão 3.2.

find(*sub*[, *start*[, *end*]])
Returns the lowest index in the object where the subsequence *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns `-1` on failure.

Alterado na versão 3.5: Writable *bytes-like object* is now accepted.

flush([*offset*[, *size*]])
Flushes changes made to the in-memory copy of a file back to disk. Without use of this call there is no guarantee that changes are written back before the object is destroyed. If *offset* and *size* are specified, only changes to the given range of bytes will be flushed to disk; otherwise, the whole extent of the mapping is flushed. *offset* must be a multiple of the `PAGESIZE` or `ALLOCATIONGRANULARITY`.

(Windows version) A nonzero value returned indicates success; zero indicates failure.

(Unix version) A zero value is returned to indicate success. An exception is raised when the call failed.

move(*dest*, *src*, *count*)
Copy the *count* bytes starting at offset *src* to the destination index *dest*. If the mmap was created with `ACCESS_READ`, then calls to move will raise a *TypeError* exception.

read([*n*])
Return a *bytes* containing up to *n* bytes starting from the current file position. If the argument is omitted, `None` or negative, return all bytes from the current file position to the end of the mapping. The file position is updated to point after the bytes that were returned.

Alterado na versão 3.3: Argument can be omitted or `None`.

read_byte()
Returns a byte at the current file position as an integer, and advances the file position by 1.

readline()
Returns a single line, starting at the current file position and up to the next newline.

resize(*newsize*)
Resizes the map and the underlying file, if any. If the mmap was created with `ACCESS_READ` or `ACCESS_COPY`, resizing the map will raise a *TypeError* exception.

rfind(*sub*[, *start*[, *end*]])
Returns the highest index in the object where the subsequence *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns `-1` on failure.

Alterado na versão 3.5: Writable *bytes-like object* is now accepted.

seek(*pos*[, *whence*])
Set the file's current position. *whence* argument is optional and defaults to `os.SEEK_SET` or `0` (absolute file

positioning); other values are `os.SEEK_CUR` or `1` (seek relative to the current position) and `os.SEEK_END` or `2` (seek relative to the file's end).

size()
Return the length of the file, which can be larger than the size of the memory-mapped area.

tell()
Returns the current position of the file pointer.

write(*bytes*)
Write the bytes in *bytes* into memory at the current position of the file pointer and return the number of bytes written (never less than `len(bytes)`, since if the write fails, a *ValueError* will be raised). The file position is updated to point after the bytes that were written. If the mmap was created with `ACCESS_READ`, then writing to it will raise a *TypeError* exception.

Alterado na versão 3.5: Writable *bytes-like object* is now accepted.

Alterado na versão 3.6: The number of bytes written is now returned.

write_byte(*byte*)
Write the integer *byte* into memory at the current position of the file pointer; the file position is advanced by `1`. If the mmap was created with `ACCESS_READ`, then writing to it will raise a *TypeError* exception.

CAPÍTULO 20

Manuseio de Dados na Internet

Este capítulo descreve módulos que suportam a manipulação de formatos de dados comumente usados na Internet.

20.1 `email` — Um email e um pacote MIME manipulável

Código Fonte: Lib/email/__init__.py

O pacote `email` é uma biblioteca para gerenciar mensagens de e-mail. Ela foi especificamente *não* projetada para enviar mensagens de e-mail para SMTP (**RFC 2821**), NNTP ou outros servidores; essas são funções de módulos como `smtplib` e `nntplib`. O pacote `email` tenta ser o mais compatível possível com RFC, suportando **RFC 5233** e **RFC 6532**, bem como os RFCs relacionados ao MIME como **RFC 2045**, **RFC 2046**, **RFC 2047**, **RFC 2183** e **RFC 2231**.

No geral a estrutura do pacote de email pode ser dividida em três componentes principais, mais um quarto componente que controla o comportamento dos outros componentes.

O componente central do pacote é um "modelo de objeto" que representa mensagens de e-mail. Uma aplicação interage com o pacote principalmente através da interface do modelo de objeto definida no submódulo `message`. A aplicação pode usar essa API para fazer perguntas sobre um e-mail existente, construir um novo e-mail ou adicionar ou remover subcomponentes de e-mail que usam a mesma interface de modelo de objeto. Ou seja, seguindo a natureza das mensagens de e-mail e seus subcomponentes MIME, o modelo de objeto de e-mail é uma estrutura em árvore de objetos que fornecem a API `EmailMessage`.

Os outros dois componentes principais do pacote são `parser` e `generator`. O analisador sintático pega a versão serializada de uma mensagem de e-mail (um fluxo de bytes) e a converte em uma árvore de objetos `EmailMessage`. O gerador pega um `EmailMessage` e o transforma novamente em um fluxo de bytes serializado. (O analisador sintático e o gerador também lidam com fluxos de caracteres de texto, mas esse uso é desencorajado, pois é muito fácil terminar com mensagens que não são válidas de uma maneira ou de outra.)

O componente de controle é o módulo `policy`. Cada `EmailMessage`, cada `generator` e cada `parser` tem um objeto associado `policy` que controla seu comportamento. Normalmente, uma aplicação precisa especificar a política apenas quando uma `EmailMessage` é criada, instanciando diretamente uma `EmailMessage` para criar um novo e-mail ou analisando um fluxo de entrada usando um `parser`. Mas a política pode ser alterada quando a mensagem é

serializada usando um *generator*. Isso permite, por exemplo, analisar uma mensagem de e-mail genérica do disco, mas serializá-la usando as configurações SMTP padrão ao enviá-la para um servidor de e-mail.

O pacote de e-mail faz o possível para ocultar os detalhes das várias RFCs em vigor da aplicação. Conceitualmente, a aplicação deve tratar a mensagem de e-mail como uma árvore estruturada de texto unicode e anexos binários, sem ter que se preocupar com a forma como eles são representados quando serializados. Na prática, no entanto, muitas vezes é necessário estar ciente de pelo menos algumas das regras que regem as mensagens MIME e sua estrutura, especificamente os nomes e a natureza dos "tipos de conteúdo" MIME e como eles identificam documentos com várias partes. Na maioria das vezes, esse conhecimento só deve ser necessário para aplicações mais complexos e, mesmo assim, deve ser apenas a estrutura de alto nível em questão, e não os detalhes de como essas estruturas são representadas. Como os tipos de conteúdo MIME são amplamente utilizados no software moderno da Internet (não apenas no e-mail), este será um conceito familiar para muitos programadores.

As seções a seguir descrevem a funcionalidade do pacote *email*. Começamos com o modelo de objeto *message*, que é a interface principal que uma aplicação usará, e seguimos com os componentes de *parser* e *generator*. Em seguida, abordamos os controles *policy*, que concluem o tratamento dos principais componentes da biblioteca.

As próximas três seções cobrem as exceções que o pacote pode apresentar e os defeitos (não conformidade com as RFCs) que o *parser* pode detectar. A seguir, abordamos os subcomponentes *headerregistry* e os subcomponentes *contentmanager*, que fornecem ferramentas para manipulação mais detalhada de cabeçalhos e cargas úteis, respectivamente. Ambos os componentes contêm recursos relevantes para consumir e produzir mensagens não triviais, mas também documentam suas APIs de extensibilidade, que serão de interesse para aplicações avançadas.

A seguir, é apresentado um conjunto de exemplos de uso das partes fundamentais das APIs abordadas nas seções anteriores.

O exposto acima representa a API moderna (compatível com unicode) do pacote de e-mail. As seções restantes, começando com a classe *Message*, cobrem a API legada *compat32* que lida muito mais diretamente com os detalhes de como as mensagens de e-mail são representadas. A API *compat32* *não* oculta os detalhes dos RFCs da aplicação, mas para aplicações que precisam operar nesse nível, eles podem ser ferramentas úteis. Esta documentação também é relevante para aplicações que ainda estão usando a API *compat32* por motivos de compatibilidade com versões anteriores.

Alterado na versão 3.6: Documentos reorganizados e reescritos para promover a nova API *EmailMessage*/*EmailPolicy*.

Conteúdos da documentação do pacote *email*:

20.1.1 `email.message`: Representing an email message

Código Fonte: Lib/email/message.py

Novo na versão 3.6:[1]

The central class in the *email* package is the *EmailMessage* class, imported from the *email.message* module. It is the base class for the *email* object model. *EmailMessage* provides the core functionality for setting and querying header fields, for accessing message bodies, and for creating or modifying structured messages.

An email message consists of *headers* and a *payload* (which is also referred to as the *content*). Headers are **RFC 5322** or **RFC 6532** style field names and values, where the field name and value are separated by a colon. The colon is not part of either the field name or the field value. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as *multipart/** or *message/rfc822*.

The conceptual model provided by an *EmailMessage* object is that of an ordered dictionary of headers coupled with a *payload* that represents the **RFC 5322** body of the message, which might be a list of sub-`EmailMessage` objects.

[1] Originally added in 3.4 as a *provisional module*. Docs for legacy message class moved to *email.message.Message: Representing an email message using the compat32 API*.

In addition to the normal dictionary methods for accessing the header names and values, there are methods for accessing specialized information from the headers (for example the MIME content type), for operating on the payload, for generating a serialized version of the message, and for recursively walking over the object tree.

The *EmailMessage* dictionary-like interface is indexed by the header names, which must be ASCII values. The values of the dictionary are strings with some extra methods. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. Unlike a real dict, there is an ordering to the keys, and there can be duplicate keys. Additional methods are provided for working with headers that have duplicate keys.

The *payload* is either a string or bytes object, in the case of simple message objects, or a list of *EmailMessage* objects, for MIME container documents such as *multipart/** and *message/rfc822* message objects.

class email.message.**EmailMessage**(*policy=default*)

If *policy* is specified use the rules it specifies to update and serialize the representation of the message. If *policy* is not set, use the *default* policy, which follows the rules of the email RFCs except for line endings (instead of the RFC mandated `\r\n`, it uses the Python standard `\n` line endings). For more information see the *policy* documentation.

as_string(*unixfrom=False*, *maxheaderlen=None*, *policy=None*)

Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. For backward compatibility with the base *Message* class *maxheaderlen* is accepted, but defaults to `None`, which means that by default the line length is controlled by the `max_line_length` of the policy. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the *Generator*.

Flattening the message may trigger changes to the *EmailMessage* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not be the most useful way to serialize messages in your application, especially if you are dealing with multiple messages. See *email.generator.Generator* for a more flexible API for serializing messages. Note also that this method is restricted to producing messages serialized as "7 bit clean" when *utf8* is `False`, which is the default.

Alterado na versão 3.6: the default behavior when *maxheaderlen* is not specified was changed from defaulting to 0 to defaulting to the value of *max_line_length* from the policy.

__str__()

Equivalent to `as_string(policy=self.policy.clone(utf8=True))`. Allows `str(msg)` to produce a string containing the serialized message in a readable format.

Alterado na versão 3.4: the method was changed to use `utf8=True`, thus producing an **RFC 6531**-like message representation, instead of being a direct alias for *as_string()*.

as_bytes(*unixfrom=False*, *policy=None*)

Return the entire message flattened as a bytes object. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the *BytesGenerator*.

Flattening the message may trigger changes to the *EmailMessage* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not be the most useful way to serialize messages in your application, especially if you are dealing with multiple messages. See *email.generator.BytesGenerator* for a more flexible API for serializing messages.

__bytes__()

Equivalent to *as_bytes()*. Allows `bytes(msg)` to produce a bytes object containing the serialized message.

is_multipart()
Return `True` if the message's payload is a list of sub-*EmailMessage* objects, otherwise return `False`. When *is_multipart()* returns `False`, the payload should be a string object (which might be a CTE encoded binary payload). Note that *is_multipart()* returning `True` does not necessarily mean that "msg.get_content_maintype() == 'multipart'" will return the `True`. For example, `is_multipart` will return `True` when the *EmailMessage* is of type `message/rfc822`.

set_unixfrom(*unixfrom*)
Set the message's envelope header to *unixfrom*, which should be a string. (See *mboxMessage* for a brief description of this header.)

get_unixfrom()
Return the message's envelope header. Defaults to `None` if the envelope header was never set.

The following methods implement the mapping-like interface for accessing the message's headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by *keys()*, but in an *EmailMessage* object, headers are always returned in the order they appeared in the original message, or in which they were added to the message later. Any header deleted and then re-added is always appended to the end of the header list.

These semantic differences are intentional and are biased toward convenience in the most common use cases.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

__len__()
Return the total number of headers, including duplicates.

__contains__(*name*)
Return `True` if the message object has a field named *name*. Matching is done without regard to case and *name* does not include the trailing colon. Used for the `in` operator. For example:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)
Return the value of the named header field. *name* does not include the colon field separator. If the header is missing, `None` is returned; a *KeyError* is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the *get_all()* method to get the values of all the extant headers named *name*.

Using the standard (non-`compat32`) policies, the returned value is an instance of a subclass of *email.headerregistry.BaseHeader*.

__setitem__(*name*, *val*)
Add a header to the message with field name *name* and value *val*. The field is appended to the end of the message's existing headers.

Note that this does *not* overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name *name*, delete the field first, e.g.:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

If the `policy` defines certain headers to be unique (as the standard policies do), this method may raise a *ValueError* when an attempt is made to assign a value to such a header when one already exists. This behavior is intentional for consistency's sake, but do not depend on it as we may choose to make such assignments do an automatic deletion of the existing header in the future.

__delitem__(*name*)
Delete all occurrences of the field with name *name* from the message's headers. No exception is raised if the named field isn't present in the headers.

keys()
Return a list of all the message's header field names.

values()
Return a list of all the message's field values.

items()
Return a list of 2-tuples containing all the message's field headers and values.

get(*name*, *failobj=None*)
Return the value of the named header field. This is identical to *__getitem__()* except that optional *failobj* is returned if the named header is missing (*failobj* defaults to `None`).

Here are some additional useful header related methods:

get_all(*name*, *failobj=None*)
Return a list of all the values for the field named *name*. If there are no such named headers in the message, *failobj* is returned (defaults to `None`).

add_header(*_name*, *_value*, ***_params*)
Extended header setting. This method is similar to *__setitem__()* except that additional header parameters can be provided as keyword arguments. *_name* is the header field to add and *_value* is the *primary* value for the header.

For each item in the keyword argument dictionary *_params*, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as `key="value"` unless the value is `None`, in which case only the key will be added.

If the value contains non-ASCII characters, the charset and language may be explicitly controlled by specifying the value as a three tuple in the format `(CHARSET, LANGUAGE, VALUE)`, where `CHARSET` is a string naming the charset to be used to encode the value, `LANGUAGE` can usually be set to `None` or the empty string (see **RFC 2231** for other possibilities), and `VALUE` is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII characters, it is automatically encoded in **RFC 2231** format using a `CHARSET` of `utf-8` and a `LANGUAGE` of `None`.

Aqui está um exemplo:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

An example of the extended interface with non-ASCII characters:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

replace_header(*_name*, *_value*)
Replace a header. Replace the first header found in the message that matches *_name*, retaining header order and field name case of the original header. If no matching header is found, raise a *KeyError*.

get_content_type()
Return the message's content type, coerced to lower case of the form *maintype/subtype*. If there is no *Content-Type* header in the message return the value returned by *get_default_type()*. If the *Content-Type* header is invalid, return `text/plain`.

(According to **RFC 2045**, messages always have a default type, *get_content_type()* will always return a value. **RFC 2045** defines a message's default type to be *text/plain* unless it appears inside a *multipart/digest* container, in which case it would be *message/rfc822*. If the *Content-Type* header has an invalid type specification, **RFC 2045** mandates that the default type be *text/plain*.)

get_content_maintype()
Return the message's main content type. This is the *maintype* part of the string returned by *get_content_type()*.

get_content_subtype()
Return the message's sub-content type. This is the *subtype* part of the string returned by *get_content_type()*.

get_default_type()
Return the default content type. Most messages have a default content type of *text/plain*, except for messages that are subparts of *multipart/digest* containers. Such subparts have a default content type of *message/rfc822*.

set_default_type(*ctype*)
Set the default content type. *ctype* should either be *text/plain* or *message/rfc822*, although this is not enforced. The default content type is not stored in the *Content-Type* header, so it only affects the return value of the `get_content_type` methods when no *Content-Type* header is present in the message.

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)
Set a parameter in the *Content-Type* header. If the parameter already exists in the header, replace its value with *value*. When *header* is `Content-Type` (the default) and the header does not yet exist in the message, add it, set its value to *text/plain*, and append the new parameter value. Optional *header* specifies an alternative header to *Content-Type*.

If the value contains non-ASCII characters, the charset and language may be explicitly specified using the optional *charset* and *language* parameters. Optional *language* specifies the **RFC 2231** language, defaulting to the empty string. Both *charset* and *language* should be strings. The default is to use the `utf8` *charset* and `None` for the *language*.

If *replace* is `False` (the default) the header is moved to the end of the list of headers. If *replace* is `True`, the header will be updated in place.

Uso do parâmetro *requote* com objetos *EmailMessage* está descontinuado.

Note that existing parameter values of headers may be accessed through the `params` attribute of the header value (for example, `msg['Content-Type'].params['charset']`).

Alterado na versão 3.4: Palavra-chave `replace` foi adicionada.

del_param(*param*, *header='content-type'*, *requote=True*)
Remove the given parameter completely from the *Content-Type* header. The header will be re-written in place without the parameter or its value. Optional *header* specifies an alternative to *Content-Type*.

Uso do parâmetro *requote* com objetos *EmailMessage* está descontinuado.

get_filename(*failobj=None*)
Return the value of the `filename` parameter of the *Content-Disposition* header of the message. If the header does not have a `filename` parameter, this method falls back to looking for the `name` parameter on the *Content-Type* header. If neither is found, or the header is missing, then *failobj* is returned. The returned string will always be unquoted as per *email.utils.unquote()*.

get_boundary(*failobj=None*)
Return the value of the `boundary` parameter of the *Content-Type* header of the message, or *failobj* if

either the header is missing, or has no `boundary` parameter. The returned string will always be unquoted as per *email.utils.unquote()*.

set_boundary(*boundary*)
Set the `boundary` parameter of the *Content-Type* header to *boundary*. *set_boundary()* will always quote *boundary* if necessary. A *HeaderParseError* is raised if the message object has no *Content-Type* header.

Note that using this method is subtly different from deleting the old *Content-Type* header and adding a new one with the new boundary via *add_header()*, because *set_boundary()* preserves the order of the *Content-Type* header in the list of headers.

get_content_charset(*failobj=None*)
Return the `charset` parameter of the *Content-Type* header, coerced to lower case. If there is no *Content-Type* header, or if that header has no `charset` parameter, *failobj* is returned.

get_charsets(*failobj=None*)
Return a list containing the character set names in the message. If the message is a *multipart*, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the `charset` parameter in the *Content-Type* header for the represented subpart. If the subpart has no *Content-Type* header, no `charset` parameter, or is not of the *text* main MIME type, then that item in the returned list will be *failobj*.

is_attachment()
Return `True` if there is a *Content-Disposition* header and its (case insensitive) value is `attachment`, `False` otherwise.

Alterado na versão 3.4.2: is_attachment is now a method instead of a property, for consistency with *is_multipart()*.

get_content_disposition()
Return the lowercased value (without parameters) of the message's *Content-Disposition* header if it has one, or `None`. The possible values for this method are *inline*, *attachment* or `None` if the message follows **RFC 2183**.

Novo na versão 3.5.

The following methods relate to interrogating and manipulating the content (payload) of the message.

walk()
The *walk()* method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use *walk()* as the iterator in a `for` loop; each iteration returns the next subpart.

Here's an example that prints the MIME type of every part of a multipart message structure:

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

`walk` iterates over the subparts of any part where *is_multipart()* returns `True`, even though `msg.get_content_maintype() == 'multipart'` may return `False`. We can see this in our example by making use of the `_structure` debug helper function:

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

Here the `message` parts are not `multiparts`, but they do contain subparts. `is_multipart()` returns `True` and `walk` descends into the subparts.

get_body (*preferencelist=('related', 'html', 'plain')*)
Return the MIME part that is the best candidate to be the "body" of the message.

preferencelist must be a sequence of strings from the set `related`, `html`, and `plain`, and indicates the order of preference for the content type of the part returned.

Start looking for candidate matches with the object on which the `get_body` method is called.

If `related` is not included in *preferencelist*, consider the root part (or subpart of the root part) of any related encountered as a candidate if the (sub-)part matches a preference.

When encountering a `multipart/related`, check the `start` parameter and if a part with a matching *Content-ID* is found, consider only it when looking for candidate matches. Otherwise consider only the first (default root) part of the `multipart/related`.

If a part has a *Content-Disposition* header, only consider the part a candidate match if the value of the header is `inline`.

If none of the candidates matches any of the preferences in *preferencelist*, return `None`.

Notes: (1) For most applications the only *preferencelist* combinations that really make sense are `('plain',)`, `('html', 'plain')`, and the default `('related', 'html', 'plain')`. (2) Because matching starts with the object on which `get_body` is called, calling `get_body` on a `multipart/related` will return the object itself unless *preferencelist* has a non-default value. (3) Messages (or message parts) that do not specify a *Content-Type* or whose *Content-Type* header is invalid will be treated as if they are of type `text/plain`, which may occasionally cause `get_body` to return unexpected results.

iter_attachments()
Return an iterator over all of the immediate sub-parts of the message that are not candidate "body" parts. That is, skip the first occurrence of each of `text/plain`, `text/html`, `multipart/related`, or `multipart/alternative` (unless they are explicitly marked as attachments via *Content-Disposition: attachment*), and return all remaining parts. When applied directly to a `multipart/related`, return an iterator over the all the related parts except the root part (ie: the part pointed to by the `start` parameter, or the first part if there is no `start` parameter or the `start` parameter doesn't match the *Content-ID* of any of the parts). When applied directly to a `multipart/alternative` or a non-`multipart`, return an empty iterator.

iter_parts()
Return an iterator over all of the immediate sub-parts of the message, which will be empty for a non-multipart. (See also *walk()*.)

get_content(**args*, *content_manager=None*, ***kw*)
Call the *get_content()* method of the *content_manager*, passing self as the message object, and passing along any other arguments or keywords as additional arguments. If *content_manager* is not specified, use the content_manager specified by the current *policy*.

set_content(**args*, *content_manager=None*, ***kw*)
Call the *set_content()* method of the *content_manager*, passing self as the message object, and passing along any other arguments or keywords as additional arguments. If *content_manager* is not specified, use the content_manager specified by the current *policy*.

make_related(*boundary=None*)
Convert a non-multipart message into a multipart/related message, moving any existing *Content-* headers and payload into a (new) first part of the multipart. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

make_alternative(*boundary=None*)
Convert a non-multipart or a multipart/related into a multipart/alternative, moving any existing *Content-* headers and payload into a (new) first part of the multipart. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

make_mixed(*boundary=None*)
Convert a non-multipart, a multipart/related, or a multipart-alternative into a multipart/mixed, moving any existing *Content-* headers and payload into a (new) first part of the multipart. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

add_related(**args*, *content_manager=None*, ***kw*)
If the message is a multipart/related, create a new message object, pass all of the arguments to its *set_content()* method, and *attach()* it to the multipart. If the message is a non-multipart, call *make_related()* and then proceed as above. If the message is any other type of multipart, raise a *TypeError*. If *content_manager* is not specified, use the content_manager specified by the current *policy*. If the added part has no *Content-Disposition* header, add one with the value inline.

add_alternative(**args*, *content_manager=None*, ***kw*)
If the message is a multipart/alternative, create a new message object, pass all of the arguments to its *set_content()* method, and *attach()* it to the multipart. If the message is a non-multipart or multipart/related, call *make_alternative()* and then proceed as above. If the message is any other type of multipart, raise a *TypeError*. If *content_manager* is not specified, use the content_manager specified by the current *policy*.

add_attachment(**args*, *content_manager=None*, ***kw*)
If the message is a multipart/mixed, create a new message object, pass all of the arguments to its *set_content()* method, and *attach()* it to the multipart. If the message is a non-multipart, multipart/related, or multipart/alternative, call *make_mixed()* and then proceed as above. If *content_manager* is not specified, use the content_manager specified by the current *policy*. If the added part has no *Content-Disposition* header, add one with the value attachment. This method can be used both for explicit attachments (*Content-Disposition: attachment*) and inline attachments (*Content-Disposition: inline*), by passing appropriate options to the content_manager.

clear()
Remove the payload and all of the headers.

clear_content()
Remove the payload and all of the Content- headers, leaving all other headers intact and in their original order.

EmailMessage objects have the following instance attributes:

preamble
The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The *preamble* attribute contains this leading extra-armor text for MIME documents. When the *Parser* discovers some text after the headers but before the first boundary string, it assigns this text to the message's *preamble* attribute. When the *Generator* is writing out the plain text representation of a MIME message, and it finds the message has a *preamble* attribute, it will write this text in the area between the headers and the first boundary. See *email.parser* and *email.generator* for details.

Note that if the message object has no preamble, the *preamble* attribute will be None.

epilogue
The *epilogue* attribute acts the same way as the *preamble* attribute, except that it contains text that appears between the last boundary and the end of the message. As with the *preamble*, if there is no epilog text this attribute will be None.

defects
The *defects* attribute contains a list of all the problems found when parsing this message. See *email.errors* for a detailed description of the possible parsing defects.

class email.message.**MIMEPart**(*policy=default*)
This class represents a subpart of a MIME message. It is identical to *EmailMessage*, except that no *MIME-Version* headers are added when *set_content()* is called, since sub-parts do not need their own *MIME-Version* headers.

20.1.2 email.parser: Parsing email messages

Código Fonte: Lib/email/parser.py

Message object structures can be created in one of two ways: they can be created from whole cloth by creating an *EmailMessage* object, adding headers using the dictionary interface, and adding payload(s) using *set_content()* and related methods, or they can be created by parsing a serialized representation of the email message.

The *email* package provides a standard parser that understands most email document structures, including MIME documents. You can pass the parser a bytes, string or file object, and the parser will return to you the root *EmailMessage* instance of the object structure. For simple, non-MIME messages the payload of this root object will likely be a string containing the text of the message. For MIME messages, the root object will return True from its *is_multipart()* method, and the subparts can be accessed via the payload manipulation methods, such as *get_body()*, *iter_parts()*, and *walk()*.

There are actually two parser interfaces available for use, the *Parser* API and the incremental *FeedParser* API. The *Parser* API is most useful if you have the entire text of the message in memory, or if the entire message lives in a file on the file system. *FeedParser* is more appropriate when you are reading the message from a stream which might block waiting for more input (such as reading an email message from a socket). The *FeedParser* can consume and parse the message incrementally, and only returns the root object when you close the parser.

Note that the parser can be extended in limited ways, and of course you can implement your own parser completely from scratch. All of the logic that connects the *email* package's bundled parser and the *EmailMessage* class is embodied

in the `policy` class, so a custom parser can create message object trees any way it finds necessary by implementing custom versions of the appropriate `policy` methods.

API do FeedParser

The `BytesFeedParser`, imported from the `email.feedparser` module, provides an API that is conducive to incremental parsing of email messages, such as would be necessary when reading the text of an email message from a source that can block (such as a socket). The `BytesFeedParser` can of course be used to parse an email message fully contained in a *bytes-like object*, string, or file, but the `BytesParser` API may be more convenient for such use cases. The semantics and results of the two parser APIs are identical.

The `BytesFeedParser`'s API is simple; you create an instance, feed it a bunch of bytes until there's no more to feed it, then close the parser to retrieve the root message object. The `BytesFeedParser` is extremely accurate when parsing standards-compliant messages, and it does a very good job of parsing non-compliant messages, providing information about how a message was deemed broken. It will populate a message object's `defects` attribute with a list of any problems it found in a message. See the `email.errors` module for the list of defects that it can find.

Here is the API for the `BytesFeedParser`:

class email.parser.**BytesFeedParser**(*_factory=None*, ***, *policy=policy.compat32*)

Create a `BytesFeedParser` instance. Optional *_factory* is a no-argument callable; if not specified use the `message_factory` from the *policy*. Call *_factory* whenever a new message object is needed.

If *policy* is specified use the rules it specifies to update the representation of the message. If *policy* is not set, use the `compat32` policy, which maintains backward compatibility with the Python 3.2 version of the email package and provides `Message` as the default factory. All other policies provide `EmailMessage` as the default *_factory*. For more information on what else *policy* controls, see the `policy` documentation.

Note: **The policy keyword should always be specified**; The default will change to `email.policy.default` in a future version of Python.

Novo na versão 3.2.

Alterado na versão 3.3: Added the *policy* keyword.

Alterado na versão 3.6: *_factory* defaults to the policy `message_factory`.

feed(*data*)

Feed the parser some more data. *data* should be a *bytes-like object* containing one or more lines. The lines can be partial and the parser will stitch such partial lines together properly. The lines can have any of the three common line endings: carriage return, newline, or carriage return and newline (they can even be mixed).

close()

Complete the parsing of all previously fed data and return the root message object. It is undefined what happens if `feed()` is called after this method has been called.

class email.parser.**FeedParser**(*_factory=None*, ***, *policy=policy.compat32*)

Works like `BytesFeedParser` except that the input to the `feed()` method must be a string. This is of limited utility, since the only way for such a message to be valid is for it to contain only ASCII text or, if `utf8` is `True`, no binary attachments.

Alterado na versão 3.3: Added the *policy* keyword.

Parser API

The `BytesParser` class, imported from the `email.parser` module, provides an API that can be used to parse a message when the complete contents of the message are available in a *bytes-like object* or file. The `email.parser` module also provides `Parser` for parsing strings, and header-only parsers, `BytesHeaderParser` and `HeaderParser`, which can be used if you're only interested in the headers of the message. `BytesHeaderParser` and `HeaderParser` can be much faster in these situations, since they do not attempt to parse the message body, instead setting the payload to the raw body.

class email.parser.**BytesParser**(*_class=None*, ***, *policy=policy.compat32*)

Create a `BytesParser` instance. The *_class* and *policy* arguments have the same meaning and semantics as the *_factory* and *policy* arguments of `BytesFeedParser`.

Note: **The policy keyword should always be specified**; The default will change to `email.policy.default` in a future version of Python.

Alterado na versão 3.3: Removed the *strict* argument that was deprecated in 2.4. Added the *policy* keyword.

Alterado na versão 3.6: *_class* defaults to the policy `message_factory`.

parse(*fp*, *headersonly=False*)

Read all the data from the binary file-like object *fp*, parse the resulting bytes, and return the message object. *fp* must support both the `readline()` and the `read()` methods.

The bytes contained in *fp* must be formatted as a block of **RFC 5322** (or, if `utf8` is `True`, **RFC 6532**) style headers and header continuation lines, optionally preceded by an envelope header. The header block is terminated either by the end of the data or by a blank line. Following the header block is the body of the message (which may contain MIME-encoded subparts, including subparts with a `Content-Transfer-Encoding` of `8bit`).

Optional *headersonly* is a flag specifying whether to stop parsing after reading the headers or not. The default is `False`, meaning it parses the entire contents of the file.

parsebytes(*bytes*, *headersonly=False*)

Similar to the `parse()` method, except it takes a *bytes-like object* instead of a file-like object. Calling this method on a *bytes-like object* is equivalent to wrapping *bytes* in a `BytesIO` instance first and calling `parse()`.

Optional *headersonly* is as with the `parse()` method.

Novo na versão 3.2.

class email.parser.**BytesHeaderParser**(*_class=None*, ***, *policy=policy.compat32*)

Exactly like `BytesParser`, except that *headersonly* defaults to `True`.

Novo na versão 3.3.

class email.parser.**Parser**(*_class=None*, ***, *policy=policy.compat32*)

This class is parallel to `BytesParser`, but handles string input.

Alterado na versão 3.3: Removed the *strict* argument. Added the *policy* keyword.

Alterado na versão 3.6: *_class* defaults to the policy `message_factory`.

parse(*fp*, *headersonly=False*)

Read all the data from the text-mode file-like object *fp*, parse the resulting text, and return the root message object. *fp* must support both the `readline()` and the `read()` methods on file-like objects.

Other than the text mode requirement, this method operates like `BytesParser.parse()`.

parsestr(*text*, *headersonly=False*)

Similar to the `parse()` method, except it takes a string object instead of a file-like object. Calling this method on a string is equivalent to wrapping *text* in a `StringIO` instance first and calling `parse()`.

Optional *headersonly* is as with the `parse()` method.

class email.parser.**HeaderParser**(*_class=None*, ***, *policy=policy.compat32*)
Exactly like `Parser`, except that *headersonly* defaults to `True`.

Since creating a message object structure from a string or a file object is such a common task, four functions are provided as a convenience. They are available in the top-level `email` package namespace.

email.**message_from_bytes**(*s*, *_class=None*, ***, *policy=policy.compat32*)
Return a message object structure from a *bytes-like object*. This is equivalent to `BytesParser().parsebytes(s)`. Optional *_class* and *policy* are interpreted as with the `BytesParser` class constructor.

Novo na versão 3.2.

Alterado na versão 3.3: Removed the *strict* argument. Added the *policy* keyword.

email.**message_from_binary_file**(*fp*, *_class=None*, ***, *policy=policy.compat32*)
Return a message object structure tree from an open binary *file object*. This is equivalent to `BytesParser().parse(fp)`. *_class* and *policy* are interpreted as with the `BytesParser` class constructor.

Novo na versão 3.2.

Alterado na versão 3.3: Removed the *strict* argument. Added the *policy* keyword.

email.**message_from_string**(*s*, *_class=None*, ***, *policy=policy.compat32*)
Return a message object structure from a string. This is equivalent to `Parser().parsestr(s)`. *_class* and *policy* are interpreted as with the `Parser` class constructor.

Alterado na versão 3.3: Removed the *strict* argument. Added the *policy* keyword.

email.**message_from_file**(*fp*, *_class=None*, ***, *policy=policy.compat32*)
Return a message object structure tree from an open *file object*. This is equivalent to `Parser().parse(fp)`. *_class* and *policy* are interpreted as with the `Parser` class constructor.

Alterado na versão 3.3: Removed the *strict* argument. Added the *policy* keyword.

Alterado na versão 3.6: *_class* defaults to the policy `message_factory`.

Here's an example of how you might use `message_from_bytes()` at an interactive Python prompt:

```
>>> import email
>>> msg = email.message_from_bytes(myBytes)
```

Additional notes

Here are some notes on the parsing semantics:

- Most non-`multipart` type messages are parsed as a single message object with a string payload. These objects will return `False` for `is_multipart()`, and `iter_parts()` will yield an empty list.
- All `multipart` type messages will be parsed as a container message object with a list of sub-message objects for their payload. The outer container message will return `True` for `is_multipart()`, and `iter_parts()` will yield a list of subparts.
- Most messages with a content type of `message/*` (such as `message/delivery-status` and `message/rfc822`) will also be parsed as container object containing a list payload of length 1. Their `is_multipart()` method will return `True`. The single element yielded by `iter_parts()` will be a sub-message object.
- Some non-standards-compliant messages may not be internally consistent about their `multipart`-edness. Such messages may have a `Content-Type` header of type `multipart`, but their `is_multipart()` method may return `False`. If such messages were parsed with the `FeedParser`, they will have an instance of the

`MultipartInvariantViolationDefect` class in their *defects* attribute list. See `email.errors` for details.

20.1.3 `email.generator`: Generating MIME documents

Código Fonte: Lib/email/generator.py

One of the most common tasks is to generate the flat (serialized) version of the email message represented by a message object structure. You will need to do this if you want to send your message via `smtplib.SMTP.sendmail()` or the `nntplib` module, or print the message on the console. Taking a message object structure and producing a serialized representation is the job of the generator classes.

As with the `email.parser` module, you aren't limited to the functionality of the bundled generator; you could write one from scratch yourself. However the bundled generator knows how to generate most email in a standards-compliant way, should handle MIME and non-MIME email messages just fine, and is designed so that the bytes-oriented parsing and generation operations are inverses, assuming the same non-transforming `policy` is used for both. That is, parsing the serialized byte stream via the `BytesParser` class and then regenerating the serialized byte stream using `BytesGenerator` should produce output identical to the input[1]. (On the other hand, using the generator on an `EmailMessage` constructed by program may result in changes to the `EmailMessage` object as defaults are filled in.)

The `Generator` class can be used to flatten a message into a text (as opposed to binary) serialized representation, but since Unicode cannot represent binary data directly, the message is of necessity transformed into something that contains only ASCII characters, using the standard email RFC Content Transfer Encoding techniques for encoding email messages for transport over channels that are not "8 bit clean".

class email.generator.**BytesGenerator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, ***, *policy=None*)

Return a `BytesGenerator` object that will write any message provided to the `flatten()` method, or any surrogateescape encoded text provided to the `write()` method, to the *file-like object* *outfp*. *outfp* must support a `write` method that accepts binary data.

If optional *mangle_from_* is `True`, put a > character in front of any line in the body that starts with the exact string `"From "`, that is `From` followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the `mangle_from_` setting of the *policy* (which is `True` for the `compat32` policy and `False` for all others). *mangle_from_* is intended for use when messages are stored in unix mbox format (see `mailbox` and WHY THE CONTENT-LENGTH FORMAT IS BAD).

If *maxheaderlen* is not `None`, refold any header lines that are longer than *maxheaderlen*, or if `0`, do not rewrap any headers. If *manheaderlen* is `None` (the default), wrap headers and other message lines according to the *policy* settings.

If *policy* is specified, use that policy to control message generation. If *policy* is `None` (the default), use the policy associated with the `Message` or `EmailMessage` object passed to `flatten` to control the message generation. See `email.policy` for details on what *policy* controls.

Novo na versão 3.2.

Alterado na versão 3.3: Added the *policy* keyword.

Alterado na versão 3.6: The default behavior of the *mangle_from_* and *maxheaderlen* parameters is to follow the policy.

[1] This statement assumes that you use the appropriate setting for `unixfrom`, and that there are no `policy` settings calling for automatic adjustments (for example, `refold_source` must be `none`, which is *not* the default). It is also not 100% true, since if the message does not conform to the RFC standards occasionally information about the exact original text is lost during parsing error recovery. It is a goal to fix these latter edge cases when possible.

flatten(*msg*, *unixfrom=False*, *linesep=None*)

Print the textual representation of the message object structure rooted at *msg* to the output file specified when the `BytesGenerator` instance was created.

If the `policy` option `cte_type` is `8bit` (the default), copy any headers in the original parsed message that have not been modified to the output with any bytes with the high bit set reproduced as in the original, and preserve the non-ASCII `Content-Transfer-Encoding` of any body parts that have them. If `cte_type` is `7bit`, convert the bytes with the high bit set as needed using an ASCII-compatible `Content-Transfer-Encoding`. That is, transform parts with non-ASCII `Content-Transfer-Encoding` (`Content-Transfer-Encoding: 8bit`) to an ASCII compatible `Content-Transfer-Encoding`, and encode RFC-invalid non-ASCII bytes in headers using the MIME `unknown-8bit` character set, thus rendering them RFC-compliant.

If *unixfrom* is `True`, print the envelope header delimiter used by the Unix mailbox format (see `mailbox`) before the first of the **RFC 5322** headers of the root message object. If the root object has no envelope header, craft a standard one. The default is `False`. Note that for subparts, no envelope header is ever printed.

If *linesep* is not `None`, use it as the separator character between all the lines of the flattened message. If *linesep* is `None` (the default), use the value specified in the *policy*.

clone(*fp*)

Return an independent clone of this `BytesGenerator` instance with the exact same option settings, and *fp* as the new *outfp*.

write(*s*)

Encode *s* using the `ASCII` codec and the `surrogateescape` error handler, and pass it to the *write* method of the *outfp* passed to the `BytesGenerator`'s constructor.

As a convenience, `EmailMessage` provides the methods `as_bytes()` and `bytes(aMessage)` (a.k.a. `__bytes__()`), which simplify the generation of a serialized binary representation of a message object. For more detail, see `email.message`.

Because strings cannot represent binary data, the `Generator` class must convert any binary data in any message it flattens to an ASCII compatible format, by converting them to an ASCII compatible `Content-Transfer_Encoding`. Using the terminology of the email RFCs, you can think of this as `Generator` serializing to an I/O stream that is not "8 bit clean". In other words, most applications will want to be using `BytesGenerator`, and not `Generator`.

class email.generator.**Generator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, ***, *policy=None*)

Return a `Generator` object that will write any message provided to the `flatten()` method, or any text provided to the `write()` method, to the *file-like object* *outfp*. *outfp* must support a `write` method that accepts string data.

If optional *mangle_from_* is `True`, put a > character in front of any line in the body that starts with the exact string `"From "`, that is `From` followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the `mangle_from_` setting of the *policy* (which is `True` for the `compat32` policy and `False` for all others). *mangle_from_* is intended for use when messages are stored in unix mbox format (see `mailbox` and WHY THE CONTENT-LENGTH FORMAT IS BAD).

If *maxheaderlen* is not `None`, refold any header lines that are longer than *maxheaderlen*, or if `0`, do not rewrap any headers. If *manheaderlen* is `None` (the default), wrap headers and other message lines according to the *policy* settings.

If *policy* is specified, use that policy to control message generation. If *policy* is `None` (the default), use the policy associated with the `Message` or `EmailMessage` object passed to `flatten` to control the message generation. See `email.policy` for details on what *policy* controls.

Alterado na versão 3.3: Added the *policy* keyword.

Alterado na versão 3.6: The default behavior of the *mangle_from_* and *maxheaderlen* parameters is to follow the policy.

flatten(*msg*, *unixfrom=False*, *linesep=None*)

Print the textual representation of the message object structure rooted at *msg* to the output file specified when the `Generator` instance was created.

If the `policy` option `cte_type` is `8bit`, generate the message as if the option were set to `7bit`. (This is required because strings cannot represent non-ASCII bytes.) Convert any bytes with the high bit set as needed using an ASCII-compatible `Content-Transfer-Encoding`. That is, transform parts with non-ASCII `Content-Transfer-Encoding` (`Content-Transfer-Encoding: 8bit`) to an ASCII compatible `Content-Transfer-Encoding`, and encode RFC-invalid non-ASCII bytes in headers using the MIME `unknown-8bit` character set, thus rendering them RFC-compliant.

If *unixfrom* is `True`, print the envelope header delimiter used by the Unix mailbox format (see `mailbox`) before the first of the **RFC 5322** headers of the root message object. If the root object has no envelope header, craft a standard one. The default is `False`. Note that for subparts, no envelope header is ever printed.

If *linesep* is not `None`, use it as the separator character between all the lines of the flattened message. If *linesep* is `None` (the default), use the value specified in the *policy*.

Alterado na versão 3.2: Added support for re-encoding `8bit` message bodies, and the *linesep* argument.

clone(*fp*)

Return an independent clone of this `Generator` instance with the exact same options, and *fp* as the new *outfp*.

write(*s*)

Write *s* to the *write* method of the *outfp* passed to the `Generator`'s constructor. This provides just enough file-like API for `Generator` instances to be used in the `print()` function.

As a convenience, `EmailMessage` provides the methods `as_string()` and `str(aMessage)` (a.k.a. `__str__()`), which simplify the generation of a formatted string representation of a message object. For more detail, see `email.message`.

The `email.generator` module also provides a derived class, `DecodedGenerator`, which is like the `Generator` base class, except that non-*text* parts are not serialized, but are instead represented in the output stream by a string derived from a template filled in with information about the part.

class email.generator.**DecodedGenerator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *fmt=None*, ***, *policy=None*)

Act like `Generator`, except that for any subpart of the message passed to `Generator.flatten()`, if the subpart is of main type *text*, print the decoded payload of the subpart, and if the main type is not *text*, instead of printing it fill in the string *fmt* using information from the part and print the resulting filled-in string.

To fill in *fmt*, execute `fmt % part_info`, where `part_info` is a dictionary composed of the following keys and values:

- `type` – Full MIME type of the non-*text* part
- `maintype` – Main MIME type of the non-*text* part
- `subtype` – Sub-MIME type of the non-*text* part
- `filename` – Filename of the non-*text* part
- `description` – Description associated with the non-*text* part
- `encoding` – Content transfer encoding of the non-*text* part

If *fmt* is `None`, use the following default *fmt*:

"[Non-text (%(type)s) part of message omitted, filename %(filename)s]"

Optional *_mangle_from_* and *maxheaderlen* are as with the `Generator` base class.

20.1.4 email.policy: Policy Objects

Novo na versão 3.3.

Código Fonte: Lib/email/policy.py

The *email* package's prime focus is the handling of email messages as described by the various email and MIME RFCs. However, the general format of email messages (a block of header fields each consisting of a name followed by a colon followed by a value, the whole block followed by a blank line and an arbitrary 'body'), is a format that has found utility outside of the realm of email. Some of these uses conform fairly closely to the main email RFCs, some do not. Even when working with email, there are times when it is desirable to break strict compliance with the RFCs, such as generating emails that interoperate with email servers that do not themselves follow the standards, or that implement extensions you want to use in ways that violate the standards.

Policy objects give the email package the flexibility to handle all these disparate use cases.

A *Policy* object encapsulates a set of attributes and methods that control the behavior of various components of the email package during use. *Policy* instances can be passed to various classes and methods in the email package to alter the default behavior. The settable values and their defaults are described below.

There is a default policy used by all classes in the email package. For all of the *parser* classes and the related convenience functions, and for the *Message* class, this is the *Compat32* policy, via its corresponding pre-defined instance *compat32*. This policy provides for complete backward compatibility (in some cases, including bug compatibility) with the pre-Python3.3 version of the email package.

This default value for the *policy* keyword to *EmailMessage* is the *EmailPolicy* policy, via its pre-defined instance *default*.

When a *Message* or *EmailMessage* object is created, it acquires a policy. If the message is created by a *parser*, a policy passed to the parser will be the policy used by the message it creates. If the message is created by the program, then the policy can be specified when it is created. When a message is passed to a *generator*, the generator uses the policy from the message by default, but you can also pass a specific policy to the generator that will override the one stored on the message object.

The default value for the *policy* keyword for the *email.parser* classes and the parser convenience functions **will be changing** in a future version of Python. Therefore you should **always specify explicitly which policy you want to use** when calling any of the classes and functions described in the *parser* module.

The first part of this documentation covers the features of *Policy*, an *abstract base class* that defines the features that are common to all policy objects, including *compat32*. This includes certain hook methods that are called internally by the email package, which a custom policy could override to obtain different behavior. The second part describes the concrete classes *EmailPolicy* and *Compat32*, which implement the hooks that provide the standard behavior and the backward compatible behavior and features, respectively.

Policy instances are immutable, but they can be cloned, accepting the same keyword arguments as the class constructor and returning a new *Policy* instance that is a copy of the original but with the specified attributes values changed.

As an example, the following code could be used to read an email message from a file on disk and pass it to the system `sendmail` program on a Unix system:

```
>>> from email import message_from_binary_file
>>> from email.generator import BytesGenerator
>>> from email import policy
>>> from subprocess import Popen, PIPE
>>> with open('mymsg.txt', 'rb') as f:
...     msg = message_from_binary_file(f, policy=policy.default)
>>> p = Popen(['sendmail', msg['To'].addresses[0]], stdin=PIPE)
>>> g = BytesGenerator(p.stdin, policy=msg.policy.clone(linesep='\r\n'))
```

(continua na próxima página)

(continuação da página anterior)

```
>>> g.flatten(msg)
>>> p.stdin.close()
>>> rc = p.wait()
```

Here we are telling *BytesGenerator* to use the RFC correct line separator characters when creating the binary string to feed into `sendmail's stdin`, where the default policy would use `\n` line separators.

Some email package methods accept a *policy* keyword argument, allowing the policy to be overridden for that method. For example, the following code uses the *as_bytes()* method of the *msg* object from the previous example and writes the message to a file using the native line separators for the platform on which it is running:

```
>>> import os
>>> with open('converted.txt', 'wb') as f:
...     f.write(msg.as_bytes(policy=msg.policy.clone(linesep=os.linesep)))
17
```

Policy objects can also be combined using the addition operator, producing a policy object whose settings are a combination of the non-default values of the summed objects:

```
>>> compat_SMTP = policy.compat32.clone(linesep='\r\n')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict_SMTP = compat_SMTP + compat_strict
```

This operation is not commutative; that is, the order in which the objects are added matters. To illustrate:

```
>>> policy100 = policy.compat32.clone(max_line_length=100)
>>> policy80 = policy.compat32.clone(max_line_length=80)
>>> apolicy = policy100 + policy80
>>> apolicy.max_line_length
80
>>> apolicy = policy80 + policy100
>>> apolicy.max_line_length
100
```

class email.policy.**Policy**(***kw*)

This is the *abstract base class* for all policy classes. It provides default implementations for a couple of trivial methods, as well as the implementation of the immutability property, the *clone()* method, and the constructor semantics.

The constructor of a policy class can be passed various keyword arguments. The arguments that may be specified are any non-method properties on this class, plus any additional non-method properties on the concrete class. A value specified in the constructor will override the default value for the corresponding attribute.

This class defines the following properties, and thus values for the following may be passed in the constructor of any policy class:

max_line_length

The maximum length of any line in the serialized output, not counting the end of line character(s). Default is 78, per **RFC 5322**. A value of `0` or *None* indicates that no line wrapping should be done at all.

linesep

The string to be used to terminate lines in serialized output. The default is `\n` because that's the internal end-of-line discipline used by Python, though `\r\n` is required by the RFCs.

cte_type

Controls the type of Content Transfer Encodings that may be or are required to be used. The possible values are:

`7bit`	all data must be "7 bit clean" (ASCII-only). This means that where necessary data will be encoded using either quoted-printable or base64 encoding.
`8bit`	data is not constrained to be 7 bit clean. Data in headers is still required to be ASCII-only and so will be encoded (see *`fold_binary()`* and *`utf8`* below for exceptions), but body parts may use the `8bit` CTE.

A `cte_type` value of `8bit` only works with `BytesGenerator`, not `Generator`, because strings cannot contain binary data. If a `Generator` is operating under a policy that specifies `cte_type=8bit`, it will act as if `cte_type` is `7bit`.

raise_on_defect
If *`True`*, any defects encountered will be raised as errors. If *`False`* (the default), defects will be passed to the *`register_defect()`* method.

mangle_from_
If *`True`*, lines starting with *"From "* in the body are escaped by putting a > in front of them. This parameter is used when the message is being serialized by a generator. Default: *`False`*.

Novo na versão 3.5: O parâmetro *mangle_from_*.

message_factory
A factory function for constructing a new empty message object. Used by the parser when building messages. Defaults to `None`, in which case *`Message`* is used.

Novo na versão 3.6.

The following *`Policy`* method is intended to be called by code using the email library to create policy instances with custom settings:

clone(***kw*)
Return a new *`Policy`* instance whose attributes have the same values as the current instance, except where those attributes are given new values by the keyword arguments.

The remaining *`Policy`* methods are called by the email package code, and are not intended to be called by an application using the email package. A custom policy must implement all of these methods.

handle_defect(*obj*, *defect*)
Handle a *defect* found on *obj*. When the email package calls this method, *defect* will always be a subclass of `Defect`.

The default implementation checks the *`raise_on_defect`* flag. If it is `True`, *defect* is raised as an exception. If it is `False` (the default), *obj* and *defect* are passed to *`register_defect()`*.

register_defect(*obj*, *defect*)
Register a *defect* on *obj*. In the email package, *defect* will always be a subclass of `Defect`.

The default implementation calls the `append` method of the `defects` attribute of *obj*. When the email package calls *`handle_defect`*, *obj* will normally have a `defects` attribute that has an `append` method. Custom object types used with the email package (for example, custom `Message` objects) should also provide such an attribute, otherwise defects in parsed messages will raise unexpected errors.

header_max_count(*name*)
Return the maximum allowed number of headers named *name*.

Called when a header is added to an *`EmailMessage`* or *`Message`* object. If the returned value is not `0` or `None`, and there are already a number of headers with the name *name* greater than or equal to the value returned, a *`ValueError`* is raised.

Because the default behavior of `Message.__setitem__` is to append the value to the list of headers, it is easy to create duplicate headers without realizing it. This method allows certain headers to be limited in

the number of instances of that header that may be added to a `Message` programmatically. (The limit is not observed by the parser, which will faithfully produce as many headers as exist in the message being parsed.)

The default implementation returns `None` for all header names.

header_source_parse(*sourcelines*)
The email package calls this method with a list of strings, each string ending with the line separation characters found in the source being parsed. The first line includes the field header name and separator. All whitespace in the source is preserved. The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the parsed header.

If an implementation wishes to retain compatibility with the existing email package policies, *name* should be the case preserved name (all characters up to the '`:`' separator), while *value* should be the unfolded value (all line separator characters removed, but whitespace kept intact), stripped of leading whitespace.

sourcelines may contain surrogateescaped binary data.

There is no default implementation

header_store_parse(*name*, *value*)
The email package calls this method with the name and value provided by the application program when the application program is modifying a `Message` programmatically (as opposed to a `Message` created by a parser). The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the header.

If an implementation wishes to retain compatibility with the existing email package policies, the *name* and *value* should be strings or string subclasses that do not change the content of the passed in arguments.

There is no default implementation

header_fetch_parse(*name*, *value*)
The email package calls this method with the *name* and *value* currently stored in the `Message` when that header is requested by the application program, and whatever the method returns is what is passed back to the application as the value of the header being retrieved. Note that there may be more than one header with the same name stored in the `Message`; the method is passed the specific name and value of the header destined to be returned to the application.

value may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the value returned by the method.

There is no default implementation

fold(*name*, *value*)
The email package calls this method with the *name* and *value* currently stored in the `Message` for a given header. The method should return a string that represents that header "folded" correctly (according to the policy settings) by composing the *name* with the *value* and inserting *linesep* characters at the appropriate places. See **RFC 5322** for a discussion of the rules for folding email headers.

value may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the string returned by the method.

fold_binary(*name*, *value*)
The same as *fold()*, except that the returned value should be a bytes object rather than a string.

value may contain surrogateescaped binary data. These could be converted back into binary data in the returned bytes object.

class email.policy.**EmailPolicy**(***kw*)
This concrete *Policy* provides behavior that is intended to be fully compliant with the current email RFCs. These include (but are not limited to) **RFC 5322**, **RFC 2047**, and the current MIME RFCs.

This policy adds new header parsing and folding algorithms. Instead of simple strings, headers are `str` subclasses with attributes that depend on the type of the field. The parsing and folding algorithm fully implement **RFC 2047** and **RFC 5322**.

The default value for the *message_factory* attribute is *EmailMessage*.

In addition to the settable attributes listed above that apply to all policies, this policy adds the following additional attributes:

Novo na versão 3.6:[1]

utf8
: If `False`, follow **RFC 5322**, supporting non-ASCII characters in headers by encoding them as "encoded words". If `True`, follow **RFC 6532** and use `utf-8` encoding for headers. Messages formatted in this way may be passed to SMTP servers that support the `SMTPUTF8` extension (**RFC 6531**).

refold_source
: If the value for a header in the `Message` object originated from a *parser* (as opposed to being set by a program), this attribute indicates whether or not a generator should refold that value when transforming the message back into serialized form. The possible values are:

`none`	all source values use original folding
`long`	source values that have any line that is longer than `max_line_length` will be refolded
`all`	todos os valores são redobrados.

O padrão é `long`.

header_factory
: A callable that takes two arguments, `name` and `value`, where `name` is a header field name and `value` is an unfolded header field value, and returns a string subclass that represents that header. A default `header_factory` (see *headerregistry*) is provided that supports custom parsing for the various address and date **RFC 5322** header field types, and the major MIME header field stypes. Support for additional custom parsing will be added in the future.

content_manager
: An object with at least two methods: get_content and set_content. When the *get_content()* or *set_content()* method of an *EmailMessage* object is called, it calls the corresponding method of this object, passing it the message object as its first argument, and any arguments or keywords that were passed to it as additional arguments. By default `content_manager` is set to *raw_data_manager*.

Novo na versão 3.4.

The class provides the following concrete implementations of the abstract methods of *Policy*:

header_max_count(*name*)
: Returns the value of the *max_count* attribute of the specialized class used to represent the header with the given name.

header_source_parse(*sourcelines*)
: The name is parsed as everything up to the '`:`' and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

header_store_parse(*name*, *value*)
: The name is returned unchanged. If the input value has a `name` attribute and it matches *name* ignoring case, the value is returned unchanged. Otherwise the *name* and *value* are passed to `header_factory`, and the resulting header object is returned as the value. In this case a `ValueError` is raised if the input value contains CR or LF characters.

[1] Originally added in 3.3 as a *provisional feature*.

header_fetch_parse(*name*, *value*)
If the value has a `name` attribute, it is returned to unmodified. Otherwise the *name*, and the *value* with any CR or LF characters removed, are passed to the `header_factory`, and the resulting header object is returned. Any surrogateescaped bytes get turned into the unicode unknown-character glyph.

fold(*name*, *value*)
Header folding is controlled by the *refold_source* policy setting. A value is considered to be a 'source value' if and only if it does not have a `name` attribute (having a `name` attribute means it is a header object of some sort). If a source value needs to be refolded according to the policy, it is converted into a header object by passing the *name* and the *value* with any CR and LF characters removed to the `header_factory`. Folding of a header object is done by calling its `fold` method with the current policy.

Source values are split into lines using *splitlines()*. If the value is not to be refolded, the lines are rejoined using the `linesep` from the policy and returned. The exception is lines containing non-ascii binary data. In that case the value is refolded regardless of the `refold_source` setting, which causes the binary data to be CTE encoded using the `unknown-8bit` charset.

fold_binary(*name*, *value*)
The same as *fold()* if *cte_type* is `7bit`, except that the returned value is bytes.

If *cte_type* is `8bit`, non-ASCII binary data is converted back into bytes. Headers with binary data are not refolded, regardless of the `refold_header` setting, since there is no way to know whether the binary data consists of single byte characters or multibyte characters.

The following instances of *EmailPolicy* provide defaults suitable for specific application domains. Note that in the future the behavior of these instances (in particular the `HTTP` instance) may be adjusted to conform even more closely to the RFCs relevant to their domains.

email.policy.**default**
An instance of `EmailPolicy` with all defaults unchanged. This policy uses the standard Python `\n` line endings rather than the RFC-correct `\r\n`.

email.policy.**SMTP**
Suitable for serializing messages in conformance with the email RFCs. Like `default`, but with `linesep` set to `\r\n`, which is RFC compliant.

email.policy.**SMTPUTF8**
The same as `SMTP` except that *utf8* is `True`. Useful for serializing messages to a message store without using encoded words in the headers. Should only be used for SMTP transmission if the sender or recipient addresses have non-ASCII characters (the *smtplib.SMTP.send_message()* method handles this automatically).

email.policy.**HTTP**
Suitable for serializing headers with for use in HTTP traffic. Like `SMTP` except that `max_line_length` is set to `None` (unlimited).

email.policy.**strict**
Convenience instance. The same as `default` except that `raise_on_defect` is set to `True`. This allows any policy to be made strict by writing:

```
somepolicy + policy.strict
```

With all of these *EmailPolicies*, the effective API of the email package is changed from the Python 3.2 API in the following ways:

- Setting a header on a *Message* results in that header being parsed and a header object created.
- Fetching a header value from a *Message* results in that header being parsed and a header object created and returned.
- Any header object, or any header that is refolded due to the policy settings, is folded using an algorithm that fully implements the RFC folding algorithms, including knowing where encoded words are required and allowed.

From the application view, this means that any header obtained through the *EmailMessage* is a header object with extra attributes, whose string value is the fully decoded unicode value of the header. Likewise, a header may be assigned a new value, or a new header created, using a unicode string, and the policy will take care of converting the unicode string into the correct RFC encoded form.

The header objects and their attributes are described in *headerregistry*.

class email.policy.**Compat32**(**kw*)
This concrete *Policy* is the backward compatibility policy. It replicates the behavior of the email package in Python 3.2. The *policy* module also defines an instance of this class, *compat32*, that is used as the default policy. Thus the default behavior of the email package is to maintain compatibility with Python 3.2.

The following attributes have values that are different from the *Policy* default:

mangle_from_
O padrão é `True`.

The class provides the following concrete implementations of the abstract methods of *Policy*:

header_source_parse(*sourcelines*)
The name is parsed as everything up to the '`:`' and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

header_store_parse(*name*, *value*)
The name and value are returned unmodified.

header_fetch_parse(*name*, *value*)
If the value contains binary data, it is converted into a *Header* object using the `unknown-8bit` charset. Otherwise it is returned unmodified.

fold(*name*, *value*)
Headers are folded using the *Header* folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. Non-ASCII binary data are CTE encoded using the `unknown-8bit` charset.

fold_binary(*name*, *value*)
Headers are folded using the *Header* folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. If `cte_type` is `7bit`, non-ascii binary data is CTE encoded using the `unknown-8bit` charset. Otherwise the original source header is used, with its existing line breaks and any (RFC invalid) binary data it may contain.

email.policy.**compat32**
An instance of *Compat32*, providing backward compatibility with the behavior of the email package in Python 3.2.

20.1.5 email.errors: Classes de Erro e Defeito.

Código Fonte: Lib/email/errors.py

A seguinte classe de exceção é definida no modulo *email.errors*.

exception email.errors.**MessageError**
Essa é a classe base para todas as exceções que o pacote *email* pode levantar. Ela é derivada da classe padão *Exception* e não define métodos adicionais.

exception email.errors.**MessageParseError**
Essa é a classe base para exceções levantadas pela classe *Parser*. Ela é derivada do *MessageError*. Essa classe pode ser usada internamente pelo analisador sintático usado pelo *headerregistry*.

exception email.errors.**HeaderParseError**
Raised under some error conditions when parsing the **RFC 5322** headers of a message, this class is derived from *MessageParseError*. The *set_boundary()* method will raise this error if the content type is unknown when the method is called. *Header* may raise this error for certain base64 decoding errors, and when an attempt is made to create a header that appears to contain an embedded header (that is, there is what is supposed to be a continuation line that has no leading whitespace and looks like a header).

exception email.errors.**BoundaryError**
Descontinuado e não mais usado.

exception email.errors.**MultipartConversionError**
Raised when a payload is added to a *Message* object using add_payload(), but the payload is already a scalar and the message's *Content-Type* main type is not either *multipart* or missing. *MultipartConversionError* multiply inherits from *MessageError* and the built-in *TypeError*.

Since Message.add_payload() is deprecated, this exception is rarely raised in practice. However the exception may also be raised if the *attach()* method is called on an instance of a class derived from *MIMENonMultipart* (e.g. *MIMEImage*).

Here is the list of the defects that the *FeedParser* can find while parsing messages. Note that the defects are added to the message where the problem was found, so for example, if a message nested inside a *multipart/alternative* had a malformed header, that nested message object would have a defect, but the containing messages would not.

All defect classes are subclassed from email.errors.MessageDefect.

- NoBoundaryInMultipartDefect – A message claimed to be a multipart, but had no *boundary* parameter.
- StartBoundaryNotFoundDefect – The start boundary claimed in the *Content-Type* header was never found.
- CloseBoundaryNotFoundDefect – A start boundary was found, but no corresponding close boundary was ever found.

  Novo na versão 3.3.
- FirstHeaderLineIsContinuationDefect – The message had a continuation line as its first header line.
- MisplacedEnvelopeHeaderDefect - A "Unix From" header was found in the middle of a header block.
- MissingHeaderBodySeparatorDefect - A line was found while parsing headers that had no leading white space but contained no ':'. Parsing continues assuming that the line represents the first line of the body.

  Novo na versão 3.3.
- MalformedHeaderDefect – A header was found that was missing a colon, or was otherwise malformed.

  Obsoleto desde a versão 3.3: This defect has not been used for several Python versions.
- MultipartInvariantViolationDefect – A message claimed to be a *multipart*, but no subparts were found. Note that when a message has this defect, its *is_multipart()* method may return False even though its content type claims to be *multipart*.
- InvalidBase64PaddingDefect – When decoding a block of base64 encoded bytes, the padding was not correct. Enough padding is added to perform the decode, but the resulting decoded bytes may be invalid.
- InvalidBase64CharactersDefect – When decoding a block of base64 encoded bytes, characters outside the base64 alphabet were encountered. The characters are ignored, but the resulting decoded bytes may be invalid.
- InvalidBase64LengthDefect – When decoding a block of base64 encoded bytes, the number of non-padding base64 characters was invalid (1 more than a multiple of 4). The encoded block was kept as-is.

20.1.6 email.headerregistry: Objetos de cabeçalho personalizados

Código Fonte: Lib/email/headerregistry.py

Novo na versão 3.6:[1]

Headers are represented by customized subclasses of *str*. The particular class used to represent a given header is determined by the *header_factory* of the *policy* in effect when the headers are created. This section documents the particular header_factory implemented by the email package for handling **RFC 5322** compliant email messages, which not only provides customized header objects for various header types, but also provides an extension mechanism for applications to add their own custom header types.

When using any of the policy objects derived from *EmailPolicy*, all headers are produced by *HeaderRegistry* and have *BaseHeader* as their last base class. Each header class has an additional base class that is determined by the type of the header. For example, many headers have the class *UnstructuredHeader* as their other base class. The specialized second class for a header is determined by the name of the header, using a lookup table stored in the *HeaderRegistry*. All of this is managed transparently for the typical application program, but interfaces are provided for modifying the default behavior for use by more complex applications.

The sections below first document the header base classes and their attributes, followed by the API for modifying the behavior of *HeaderRegistry*, and finally the support classes used to represent the data parsed from structured headers.

class email.headerregistry.**BaseHeader**(*name*, *value*)

name and *value* are passed to BaseHeader from the *header_factory* call. The string value of any header object is the *value* fully decoded to unicode.

This base class defines the following read-only properties:

name

The name of the header (the portion of the field before the ':'). This is exactly the value passed in the *header_factory* call for *name*; that is, case is preserved.

defects

A tuple of HeaderDefect instances reporting any RFC compliance problems found during parsing. The email package tries to be complete about detecting compliance issues. See the *errors* module for a discussion of the types of defects that may be reported.

max_count

The maximum number of headers of this type that can have the same name. A value of None means unlimited. The BaseHeader value for this attribute is None; it is expected that specialized header classes will override this value as needed.

BaseHeader also provides the following method, which is called by the email library code and should not in general be called by application programs:

fold(*, *policy*)

Return a string containing *linesep* characters as required to correctly fold the header according to *policy*. A *cte_type* of 8bit will be treated as if it were 7bit, since headers may not contain arbitrary binary data. If *utf8* is False, non-ASCII data will be **RFC 2047** encoded.

BaseHeader by itself cannot be used to create a header object. It defines a protocol that each specialized header cooperates with in order to produce the header object. Specifically, BaseHeader requires that the specialized class provide a *classmethod()* named parse. This method is called as follows:

```
parse(string, kwds)
```

[1] Originally added in 3.3 as a *provisional module*

kwds is a dictionary containing one pre-initialized key, defects. defects is an empty list. The parse method should append any detected defects to this list. On return, the kwds dictionary *must* contain values for at least the keys decoded and defects. decoded should be the string value for the header (that is, the header value fully decoded to unicode). The parse method should assume that *string* may contain content-transfer-encoded parts, but should correctly handle all valid unicode characters as well so that it can parse un-encoded header values.

BaseHeader's __new__ then creates the header instance, and calls its init method. The specialized class only needs to provide an init method if it wishes to set additional attributes beyond those provided by BaseHeader itself. Such an init method should look like this:

```
def init(self, *args, **kw):
    self._myattr = kw.pop('myattr')
    super().init(*args, **kw)
```

That is, anything extra that the specialized class puts in to the kwds dictionary should be removed and handled, and the remaining contents of kw (and args) passed to the BaseHeader init method.

class email.headerregistry.**UnstructuredHeader**

An "unstructured" header is the default type of header in **RFC 5322**. Any header that does not have a specified syntax is treated as unstructured. The classic example of an unstructured header is the *Subject* header.

In **RFC 5322**, an unstructured header is a run of arbitrary text in the ASCII character set. **RFC 2047**, however, has an **RFC 5322** compatible mechanism for encoding non-ASCII text as ASCII characters within a header value. When a *value* containing encoded words is passed to the constructor, the UnstructuredHeader parser converts such encoded words into unicode, following the **RFC 2047** rules for unstructured text. The parser uses heuristics to attempt to decode certain non-compliant encoded words. Defects are registered in such cases, as well as defects for issues such as invalid characters within the encoded words or the non-encoded text.

This header type provides no additional attributes.

class email.headerregistry.**DateHeader**

RFC 5322 specifies a very specific format for dates within email headers. The DateHeader parser recognizes that date format, as well as recognizing a number of variant forms that are sometimes found "in the wild".

This header type provides the following additional attributes:

datetime

If the header value can be recognized as a valid date of one form or another, this attribute will contain a *datetime* instance representing that date. If the timezone of the input date is specified as -0000 (indicating it is in UTC but contains no information about the source timezone), then *datetime* will be a naive *datetime*. If a specific timezone offset is found (including *+0000*), then *datetime* will contain an aware datetime that uses *datetime.timezone* to record the timezone offset.

The decoded value of the header is determined by formatting the datetime according to the **RFC 5322** rules; that is, it is set to:

```
email.utils.format_datetime(self.datetime)
```

When creating a DateHeader, *value* may be *datetime* instance. This means, for example, that the following code is valid and does what one would expect:

```
msg['Date'] = datetime(2011, 7, 15, 21)
```

Because this is a naive datetime it will be interpreted as a UTC timestamp, and the resulting value will have a timezone of -0000. Much more useful is to use the *localtime()* function from the *utils* module:

```
msg['Date'] = utils.localtime()
```

This example sets the date header to the current time and date using the current timezone offset.

class email.headerregistry.**AddressHeader**
Address headers are one of the most complex structured header types. The AddressHeader class provides a generic interface to any address header.

This header type provides the following additional attributes:

groups
A tuple of *Group* objects encoding the addresses and groups found in the header value. Addresses that are not part of a group are represented in this list as single-address Groups whose *display_name* is None.

addresses
A tuple of *Address* objects encoding all of the individual addresses from the header value. If the header value contains any groups, the individual addresses from the group are included in the list at the point where the group occurs in the value (that is, the list of addresses is "flattened" into a one dimensional list).

The decoded value of the header will have all encoded words decoded to unicode. *idna* encoded domain names are also decoded to unicode. The decoded value is set by *join*ing the *str* value of the elements of the groups attribute with ', '.

A list of *Address* and *Group* objects in any combination may be used to set the value of an address header. Group objects whose display_name is None will be interpreted as single addresses, which allows an address list to be copied with groups intact by using the list obtained from the groups attribute of the source header.

class email.headerregistry.**SingleAddressHeader**
Uma subclasse de *AddressHeader* que adiciona um atributo adicional:

address
The single address encoded by the header value. If the header value actually contains more than one address (which would be a violation of the RFC under the default *policy*), accessing this attribute will result in a *ValueError*.

Many of the above classes also have a Unique variant (for example, UniqueUnstructuredHeader). The only difference is that in the Unique variant, *max_count* is set to 1.

class email.headerregistry.**MIMEVersionHeader**
There is really only one valid value for the *MIME-Version* header, and that is 1.0. For future proofing, this header class supports other valid version numbers. If a version number has a valid value per **RFC 2045**, then the header object will have non-None values for the following attributes:

version
The version number as a string, with any whitespace and/or comments removed.

major
The major version number as an integer

minor
The minor version number as an integer

class email.headerregistry.**ParameterizedMIMEHeader**
MIME headers all start with the prefix 'Content-'. Each specific header has a certain value, described under the class for that header. Some can also take a list of supplemental parameters, which have a common format. This class serves as a base for all the MIME headers that take parameters.

params
A dictionary mapping parameter names to parameter values.

class email.headerregistry.**ContentTypeHeader**
A *ParameterizedMIMEHeader* class that handles the *Content-Type* header.

content_type
The content type string, in the form maintype/subtype.

maintype

subtype

class email.headerregistry.**ContentDispositionHeader**
Uma classe *ParameterizedMIMEHeader* que lida com o cabeçalho *Content-Disposition*.

content-disposition
inline and attachment are the only valid values in common use.

class email.headerregistry.**ContentTransferEncoding**
Handles the *Content-Transfer-Encoding* header.

cte
Valid values are 7bit, 8bit, base64, and quoted-printable. See **RFC 2045** for more information.

class email.headerregistry.**HeaderRegistry**(*base_class=BaseHeader*, *default_class=UnstructuredHeader*, *use_default_map=True*)
This is the factory used by *EmailPolicy* by default. HeaderRegistry builds the class used to create a header instance dynamically, using *base_class* and a specialized class retrieved from a registry that it holds. When a given header name does not appear in the registry, the class specified by *default_class* is used as the specialized class. When *use_default_map* is True (the default), the standard mapping of header names to classes is copied in to the registry during initialization. *base_class* is always the last class in the generated class's __bases__ list.

The default mappings are:

subject UniqueUnstructuredHeader

date UniqueDateHeader

resent-date DateHeader

orig-date UniqueDateHeader

sender UniqueSingleAddressHeader

resent-sender SingleAddressHeader

para UniqueAddressHeader

resent-to AddressHeader

cc UniqueAddressHeader

resent-cc AddressHeader

de UniqueAddressHeader

resent-from AddressHeader

reply-to UniqueAddressHeader

HeaderRegistry has the following methods:

map_to_type(*self*, *name*, *cls*)
name is the name of the header to be mapped. It will be converted to lower case in the registry. *cls* is the specialized class to be used, along with *base_class*, to create the class used to instantiate headers that match *name*.

__getitem__(*name*)
Construct and return a class to handle creating a *name* header.

__call__(*name*, *value*)
Retrieves the specialized header associated with *name* from the registry (using *default_class* if *name* does

not appear in the registry) and composes it with *base_class* to produce a class, calls the constructed class's constructor, passing it the same argument list, and finally returns the class instance created thereby.

The following classes are the classes used to represent data parsed from structured headers and can, in general, be used by an application program to construct structured values to assign to specific headers.

class email.headerregistry.**Address**(*display_name=''*, *username=''*, *domain=''*, *addr_spec=None*)

The class used to represent an email address. The general form of an address is:

```
[display_name] <username@domain>
```

or:

```
username@domain
```

where each part must conform to specific syntax rules spelled out in **RFC 5322**.

As a convenience *addr_spec* can be specified instead of *username* and *domain*, in which case *username* and *domain* will be parsed from the *addr_spec*. An *addr_spec* must be a properly RFC quoted string; if it is not `Address` will raise an error. Unicode characters are allowed and will be property encoded when serialized. However, per the RFCs, unicode is *not* allowed in the username portion of the address.

display_name
The display name portion of the address, if any, with all quoting removed. If the address does not have a display name, this attribute will be an empty string.

username
The `username` portion of the address, with all quoting removed.

domain
The `domain` portion of the address.

addr_spec
The `username@domain` portion of the address, correctly quoted for use as a bare address (the second form shown above). This attribute is not mutable.

__str__()
The `str` value of the object is the address quoted according to **RFC 5322** rules, but with no Content Transfer Encoding of any non-ASCII characters.

To support SMTP (**RFC 5321**), `Address` handles one special case: if `username` and `domain` are both the empty string (or `None`), then the string value of the `Address` is <>.

class email.headerregistry.**Group**(*display_name=None*, *addresses=None*)

The class used to represent an address group. The general form of an address group is:

```
display_name: [address-list];
```

As a convenience for processing lists of addresses that consist of a mixture of groups and single addresses, a `Group` may also be used to represent single addresses that are not part of a group by setting *display_name* to `None` and providing a list of the single address as *addresses*.

display_name
The `display_name` of the group. If it is `None` and there is exactly one `Address` in `addresses`, then the `Group` represents a single address that is not in a group.

addresses
A possibly empty tuple of *Address* objects representing the addresses in the group.

__str__()
The `str` value of a `Group` is formatted according to **RFC 5322**, but with no Content Transfer Encoding of any non-ASCII characters. If `display_name` is none and there is a single `Address` in the `addresses` list, the `str` value will be the same as the `str` of that single `Address`.

20.1.7 email.contentmanager: Managing MIME Content

Código Fonte: Lib/email/contentmanager.py

Novo na versão 3.6:[1]

class email.contentmanager.**ContentManager**
Base class for content managers. Provides the standard registry mechanisms to register converters between MIME content and other representations, as well as the `get_content` and `set_content` dispatch methods.

get_content(*msg*, **args*, ***kw*)
Look up a handler function based on the `mimetype` of *msg* (see next paragraph), call it, passing through all arguments, and return the result of the call. The expectation is that the handler will extract the payload from *msg* and return an object that encodes information about the extracted data.

To find the handler, look for the following keys in the registry, stopping with the first one found:

- the string representing the full MIME type (`maintype/subtype`)
- the string representing the `maintype`
- the empty string

If none of these keys produce a handler, raise a *KeyError* for the full MIME type.

set_content(*msg*, *obj*, **args*, ***kw*)
If the `maintype` is `multipart`, raise a *TypeError*; otherwise look up a handler function based on the type of *obj* (see next paragraph), call *clear_content()* on the *msg*, and call the handler function, passing through all arguments. The expectation is that the handler will transform and store *obj* into *msg*, possibly making other changes to *msg* as well, such as adding various MIME headers to encode information needed to interpret the stored data.

To find the handler, obtain the type of *obj* (`typ = type(obj)`), and look for the following keys in the registry, stopping with the first one found:

- the type itself (`typ`)
- the type's fully qualified name (`typ.__module__ + '.' + typ.__qualname__`).
- the type's qualname (`typ.__qualname__`)
- the type's name (`typ.__name__`).

If none of the above match, repeat all of the checks above for each of the types in the *MRO* (`typ.__mro__`). Finally, if no other key yields a handler, check for a handler for the key `None`. If there is no handler for `None`, raise a *KeyError* for the fully qualified name of the type.

Also add a *MIME-Version* header if one is not present (see also *MIMEPart*).

add_get_handler(*key*, *handler*)
Record the function *handler* as the handler for *key*. For the possible values of *key*, see *get_content()*.

[1] Originally added in 3.4 as a *provisional module*

add_set_handler(*typekey*, *handler*)
Record *handler* as the function to call when an object of a type matching *typekey* is passed to *set_content()*. For the possible values of *typekey*, see *set_content()*.

Content Manager Instances

Currently the email package provides only one concrete content manager, *raw_data_manager*, although more may be added in the future. *raw_data_manager* is the *content_manager* provided by *EmailPolicy* and its derivatives.

email.contentmanager.**raw_data_manager**
This content manager provides only a minimum interface beyond that provided by *Message* itself: it deals only with text, raw byte strings, and *Message* objects. Nevertheless, it provides significant advantages compared to the base API: `get_content` on a text part will return a unicode string without the application needing to manually decode it, `set_content` provides a rich set of options for controlling the headers added to a part and controlling the content transfer encoding, and it enables the use of the various `add_` methods, thereby simplifying the creation of multipart messages.

email.contentmanager.**get_content**(*msg*, *errors='replace'*)
Return the payload of the part as either a string (for `text` parts), an *EmailMessage* object (for `message/rfc822` parts), or a `bytes` object (for all other non-multipart types). Raise a *KeyError* if called on a `multipart`. If the part is a `text` part and *errors* is specified, use it as the error handler when decoding the payload to unicode. The default error handler is `replace`.

email.contentmanager.**set_content**(*msg*, *<'str'>*, *subtype="plain"*, *charset='utf-8'* *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)
email.contentmanager.**set_content**(*msg*, *<'bytes'>*, *maintype*, *subtype*, *cte="base64"*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)
email.contentmanager.**set_content**(*msg*, *<'EmailMessage'>*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)
Adicione headers e payload à *msg*:

Add a *Content-Type* header with a `maintype/subtype` value.

- For `str`, set the MIME `maintype` to `text`, and set the subtype to *subtype* if it is specified, or `plain` if it is not.
- For `bytes`, use the specified *maintype* and *subtype*, or raise a *TypeError* if they are not specified.
- For *EmailMessage* objects, set the maintype to `message`, and set the subtype to *subtype* if it is specified or `rfc822` if it is not. If *subtype* is `partial`, raise an error (`bytes` objects must be used to construct `message/partial` parts).

If *charset* is provided (which is valid only for `str`), encode the string to bytes using the specified character set. The default is `utf-8`. If the specified *charset* is a known alias for a standard MIME charset name, use the standard charset instead.

If *cte* is set, encode the payload using the specified content transfer encoding, and set the *Content-Transfer-Encoding* header to that value. Possible values for *cte* are `quoted-printable`, `base64`, `7bit`, `8bit`, and `binary`. If the input cannot be encoded in the specified encoding (for example, specifying a *cte* of `7bit` for an input that contains non-ASCII values), raise a *ValueError*.

- For `str` objects, if *cte* is not set use heuristics to determine the most compact encoding.
- For *EmailMessage*, per **RFC 2046**, raise an error if a *cte* of `quoted-printable` or `base64` is requested for *subtype* `rfc822`, and for any *cte* other than `7bit` for *subtype* `external-body`. For

`message/rfc822`, use `8bit` if *cte* is not specified. For all other values of *subtype*, use `7bit`.

Nota: A *cte* of `binary` does not actually work correctly yet. The `EmailMessage` object as modified by `set_content` is correct, but *`BytesGenerator`* does not serialize it correctly.

If *disposition* is set, use it as the value of the *`Content-Disposition`* header. If not specified, and *filename* is specified, add the header with the value `attachment`. If *disposition* is not specified and *filename* is also not specified, do not add the header. The only valid values for *disposition* are `attachment` and `inline`.

If *filename* is specified, use it as the value of the `filename` parameter of the *`Content-Disposition`* header.

If *cid* is specified, add a *`Content-ID`* header with *cid* as its value.

If *params* is specified, iterate its `items` method and use the resulting `(key, value)` pairs to set additional parameters on the *`Content-Type`* header.

If *headers* is specified and is a list of strings of the form `headername: headervalue` or a list of `header` objects (distinguished from strings by having a `name` attribute), add the headers to *msg*.

20.1.8 `email`: Exemplos

Here are a few examples of how to use the *`email`* package to read, write, and send simple email messages, as well as more complex MIME messages.

First, let's see how to create and send a simple text message (both the text content and the addresses may contain unicode characters):

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.message import EmailMessage

# Open the plain text file whose name is in textfile for reading.
with open(textfile) as fp:
    # Create a text/plain message
    msg = EmailMessage()
    msg.set_content(fp.read())

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = f'The contents of {textfile}'
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server.
s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```

Parsing **RFC 822** headers can easily be done by the using the classes from the *`parser`* module:

```
# Import the email modules we'll need
from email.parser import BytesParser, Parser
```

(continua na próxima página)

(continuação da página anterior)

```
from email.policy import default

# If the e-mail headers are in a file, uncomment these two lines:
# with open(messagefile, 'rb') as fp:
#     headers = BytesParser(policy=default).parse(fp)

#  Or for parsing headers in a string (this is an uncommon operation), use:
headers = Parser(policy=default).parsestr(
        'From: Foo Bar <user@example.com>\n'
        'To: <someone_else@example.com>\n'
        'Subject: Test message\n'
        '\n'
        'Body would go here\n')

#  Now the header items can be accessed as a dictionary:
print('To: {}'.format(headers['to']))
print('From: {}'.format(headers['from']))
print('Subject: {}'.format(headers['subject']))

# You can also access the parts of the addresses:
print('Recipient username: {}'.format(headers['to'].addresses[0].username))
print('Sender name: {}'.format(headers['from'].addresses[0].display_name))
```

Here's an example of how to send a MIME message containing a bunch of family pictures that may be residing in a directory:

```
# Import smtplib for the actual sending function
import smtplib

# And imghdr to find the types of our images
import imghdr

# Here are the email package modules we'll need
from email.message import EmailMessage

# Create the container email message.
msg = EmailMessage()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = ', '.join(family)
msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

# Open the files in binary mode.  Use imghdr to figure out the
# MIME subtype for each specific image.
for file in pngfiles:
    with open(file, 'rb') as fp:
        img_data = fp.read()
    msg.add_attachment(img_data, maintype='image',
                               subtype=imghdr.what(None, img_data))

# Send the email via our own SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

Here's an example of how to send the entire contents of a directory as an email message:[1]

```
#!/usr/bin/env python3

"""Send the contents of a directory as a MIME message."""

import os
import smtplib
# For guessing MIME type based on file name extension
import mimetypes

from argparse import ArgumentParser

from email.message import EmailMessage
from email.policy import SMTP


def main():
    parser = ArgumentParser(description="""\
Send the contents of a directory as a MIME message.
Unless the -o option is given, the email is sent by forwarding to your local
SMTP server, which then does the normal delivery process.  Your local machine
must be running an SMTP server.
""")
    parser.add_argument('-d', '--directory',
                        help="""Mail the contents of the specified directory,
                        otherwise use the current directory.  Only the regular
                        files in the directory are sent, and we don't recurse to
                        subdirectories.""")
    parser.add_argument('-o', '--output',
                        metavar='FILE',
                        help="""Print the composed message to FILE instead of
                        sending the message to the SMTP server.""")
    parser.add_argument('-s', '--sender', required=True,
                        help='The value of the From: header (required)')
    parser.add_argument('-r', '--recipient', required=True,
                        action='append', metavar='RECIPIENT',
                        default=[], dest='recipients',
                        help='A To: header value (at least one required)')
    args = parser.parse_args()
    directory = args.directory
    if not directory:
        directory = '.'
    # Create the message
    msg = EmailMessage()
    msg['Subject'] = f'Contents of directory {os.path.abspath(directory)}'
    msg['To'] = ', '.join(args.recipients)
    msg['From'] = args.sender
    msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # Guess the content type based on the file's extension.  Encoding
        # will be ignored, although we should check for simple things like
```

(continua na próxima página)

[1] Thanks to Matthew Dixon Cowles for the original inspiration and examples.

(continuação da página anterior)

```
        # gzip'd or compressed files.
        ctype, encoding = mimetypes.guess_type(path)
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded (compressed), so
            # use a generic bag-of-bits type.
            ctype = 'application/octet-stream'
        maintype, subtype = ctype.split('/', 1)
        with open(path, 'rb') as fp:
            msg.add_attachment(fp.read(),
                               maintype=maintype,
                               subtype=subtype,
                               filename=filename)
    # Now send or store the message
    if args.output:
        with open(args.output, 'wb') as fp:
            fp.write(msg.as_bytes(policy=SMTP))
    else:
        with smtplib.SMTP('localhost') as s:
            s.send_message(msg)


if __name__ == '__main__':
    main()
```

Aqui está um exemplo de como descompactar uma mensagem MIME, como a acima, para um diretório de arquivos:

```
#!/usr/bin/env python3

"""Unpack a MIME message into a directory of files."""

import os
import email
import mimetypes

from email.policy import default

from argparse import ArgumentParser


def main():
    parser = ArgumentParser(description="""\
Unpack a MIME message into a directory of files.
""")
    parser.add_argument('-d', '--directory', required=True,
                        help="""Unpack the MIME message into the named
                        directory, which will be created if it doesn't already
                        exist.""")
    parser.add_argument('msgfile')
    args = parser.parse_args()

    with open(args.msgfile, 'rb') as fp:
        msg = email.message_from_binary_file(fp, policy=default)

    try:
        os.mkdir(args.directory)
    except FileExistsError:
        pass
```

(continua na próxima página)

(continuação da página anterior)

```
    counter = 1
    for part in msg.walk():
        # multipart/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given filename so that an
        # email message can't be used to overwrite important files
        filename = part.get_filename()
        if not filename:
            ext = mimetypes.guess_extension(part.get_content_type())
            if not ext:
                # Use a generic bag-of-bits extension
                ext = '.bin'
            filename = f'part-{counter:03d}{ext}'
        counter += 1
        with open(os.path.join(args.directory, filename), 'wb') as fp:
            fp.write(part.get_payload(decode=True))


if __name__ == '__main__':
    main()
```

Here's an example of how to create an HTML message with an alternative plain text version. To make things a bit more interesting, we include a related image in the html part, and we save a copy of what we are going to send to disk, as well as sending it.

```
#!/usr/bin/env python3

import smtplib

from email.message import EmailMessage
from email.headerregistry import Address
from email.utils import make_msgid

# Create the base text message.
msg = EmailMessage()
msg['Subject'] = "Ayons asperges pour le déjeuner"
msg['From'] = Address("Pepé Le Pew", "pepe", "example.com")
msg['To'] = (Address("Penelope Pussycat", "penelope", "example.com"),
             Address("Fabrette Pussycat", "fabrette", "example.com"))
msg.set_content("""\
Salut!

Cela ressemble à un excellent recipie[1] déjeuner.

[1] http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718

--Pepé
""")

# Add the html version.  This converts the message into a multipart/alternative
# container, with the original text message as the first part and the new html
# message as the second part.
asparagus_cid = make_msgid()
msg.add_alternative("""\
```

(continua na próxima página)

(continuação da página anterior)

```
<html>
  <head></head>
  <body>
    <p>Salut!</p>
    <p>Cela ressemble à un excellent
        <a href="http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718">
            recipie
        </a> déjeuner.
    </p>
    <img src="cid:{asparagus_cid}" />
  </body>
</html>
""".format(asparagus_cid=asparagus_cid[1:-1]), subtype='html')
# note that we needed to peel the <> off the msgid for use in the html.

# Now add the related image to the html part.
with open("roasted-asparagus.jpg", 'rb') as img:
    msg.get_payload()[1].add_related(img.read(), 'image', 'jpeg',
                                     cid=asparagus_cid)

# Make a local copy of what we are going to send.
with open('outgoing.msg', 'wb') as f:
    f.write(bytes(msg))

# Send the message via local SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

If we were sent the message from the last example, here is one way we could process it:

```
import os
import sys
import tempfile
import mimetypes
import webbrowser

# Import the email modules we'll need
from email import policy
from email.parser import BytesParser

# An imaginary module that would make this work and be safe.
from imaginary import magic_html_parser

# In a real program you'd get the filename from the arguments.
with open('outgoing.msg', 'rb') as fp:
    msg = BytesParser(policy=policy.default).parse(fp)

# Now the header items can be accessed as a dictionary, and any non-ASCII will
# be converted to unicode:
print('To:', msg['to'])
print('From:', msg['from'])
print('Subject:', msg['subject'])

# If we want to print a preview of the message content, we can extract whatever
# the least formatted payload is and print the first three lines.  Of course,
# if the message has no plain text part printing the first three lines of html
# is probably useless, but this is just a conceptual example.
```

(continua na próxima página)

(continuação da página anterior)

```
simplest = msg.get_body(preferencelist=('plain', 'html'))
print()
print(''.join(simplest.get_content().splitlines(keepends=True)[:3]))

ans = input("View full message?")
if ans.lower()[0] == 'n':
    sys.exit()

# We can extract the richest alternative in order to display it:
richest = msg.get_body()
partfiles = {}
if richest['content-type'].maintype == 'text':
    if richest['content-type'].subtype == 'plain':
        for line in richest.get_content().splitlines():
            print(line)
        sys.exit()
    elif richest['content-type'].subtype == 'html':
        body = richest
    else:
        print("Don't know how to display {}".format(richest.get_content_type()))
        sys.exit()
elif richest['content-type'].content_type == 'multipart/related':
    body = richest.get_body(preferencelist=('html'))
    for part in richest.iter_attachments():
        fn = part.get_filename()
        if fn:
            extension = os.path.splitext(part.get_filename())[1]
        else:
            extension = mimetypes.guess_extension(part.get_content_type())
        with tempfile.NamedTemporaryFile(suffix=extension, delete=False) as f:
            f.write(part.get_content())
            # again strip the <> to go from email form of cid to html form.
            partfiles[part['content-id'][1:-1]] = f.name
else:
    print("Don't know how to display {}".format(richest.get_content_type()))
    sys.exit()
with tempfile.NamedTemporaryFile(mode='w', delete=False) as f:
    # The magic_html_parser has to rewrite the href="cid:...." attributes to
    # point to the filenames in partfiles.  It also has to do a safety-sanitize
    # of the html.  It could be written using html.parser.
    f.write(magic_html_parser(body.get_content(), partfiles))
webbrowser.open(f.name)
os.remove(f.name)
for fn in partfiles.values():
    os.remove(fn)

# Of course, there are lots of email messages that could break this simple
# minded program, but it will handle the most common ones.
```

Up to the prompt, the output from the above is:

```
To: Penelope Pussycat <penelope@example.com>, Fabrette Pussycat <fabrette@example.com>
From: Pepé Le Pew <pepe@example.com>
Subject: Ayons asperges pour le déjeuner

Salut!
```

(continua na próxima página)

(continuação da página anterior)

```
Cela ressemble à un excellent recipie[1] déjeuner.
```

API legada

20.1.9 email.message.Message: Representing an email message using the compat32 API

The *Message* class is very similar to the *EmailMessage* class, without the methods added by that class, and with the default behavior of certain other methods being slightly different. We also document here some methods that, while supported by the *EmailMessage* class, are not recommended unless you are dealing with legacy code.

The philosophy and structure of the two classes is otherwise the same.

This document describes the behavior under the default (for *Message*) policy *Compat32*. If you are going to use another policy, you should be using the *EmailMessage* class instead.

An email message consists of *headers* and a *payload*. Headers must be **RFC 5233** style names and values, where the field name and value are separated by a colon. The colon is not part of either the field name or the field value. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as *multipart/** or *message/rfc822*.

The conceptual model provided by a *Message* object is that of an ordered dictionary of headers with additional methods for accessing both specialized information from the headers, for accessing the payload, for generating a serialized version of the message, and for recursively walking over the object tree. Note that duplicate headers are supported but special methods must be used to access them.

The *Message* pseudo-dictionary is indexed by the header names, which must be ASCII values. The values of the dictionary are strings that are supposed to contain only ASCII characters; there is some special handling for non-ASCII input, but it doesn't always produce the correct results. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. There may also be a single envelope header, also known as the *Unix-From* header or the `From_` header. The *payload* is either a string or bytes, in the case of simple message objects, or a list of *Message* objects, for MIME container documents (e.g. *multipart/** and *message/rfc822*).

Here are the methods of the *Message* class:

class email.message.**Message**(*policy=compat32*)

If *policy* is specified (it must be an instance of a *policy* class) use the rules it specifies to update and serialize the representation of the message. If *policy* is not set, use the *compat32* policy, which maintains backward compatibility with the Python 3.2 version of the email package. For more information see the *policy* documentation.

Alterado na versão 3.3: The *policy* keyword argument was added.

as_string(*unixfrom=False*, *maxheaderlen=0*, *policy=None*)

Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. For backward compatibility reasons, *maxheaderlen* defaults to `0`, so if you want a different value you must override it explicitly (the value specified for *max_line_length* in the policy will be ignored by this method). The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the `Generator`.

Flattening the message may trigger changes to the *Message* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the

unix mbox format. For more flexibility, instantiate a *Generator* instance and use its *flatten()* method directly. For example:

```
from io import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

If the message object contains binary data that is not encoded according to RFC standards, the non-compliant data will be replaced by unicode "unknown character" code points. (See also *as_bytes()* and *BytesGenerator*.)

Alterado na versão 3.4: the *policy* keyword argument was added.

__str__()
Equivalent to *as_string()*. Allows `str(msg)` to produce a string containing the formatted message.

as_bytes(*unixfrom=False*, *policy=None*)
Return the entire message flattened as a bytes object. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the `BytesGenerator`.

Flattening the message may trigger changes to the *Message* if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the unix mbox format. For more flexibility, instantiate a *BytesGenerator* instance and use its *flatten()* method directly. For example:

```
from io import BytesIO
from email.generator import BytesGenerator
fp = BytesIO()
g = BytesGenerator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

Novo na versão 3.4.

__bytes__()
Equivalent to *as_bytes()*. Allows `bytes(msg)` to produce a bytes object containing the formatted message.

Novo na versão 3.4.

is_multipart()
Return `True` if the message's payload is a list of sub-*Message* objects, otherwise return `False`. When *is_multipart()* returns `False`, the payload should be a string object (which might be a CTE encoded binary payload). (Note that *is_multipart()* returning `True` does not necessarily mean that "msg.get_content_maintype() == 'multipart'" will return the `True`. For example, `is_multipart` will return `True` when the *Message* is of type `message/rfc822`.)

set_unixfrom(*unixfrom*)
Set the message's envelope header to *unixfrom*, which should be a string.

get_unixfrom()
Return the message's envelope header. Defaults to `None` if the envelope header was never set.

attach(*payload*)
Add the given *payload* to the current payload, which must be `None` or a list of *Message* objects before the call. After the call, the payload will always be a list of *Message* objects. If you want to set the payload to a scalar object (e.g. a string), use *set_payload()* instead.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by *set_content()* and the related `make` and `add` methods.

get_payload(*i=None*, *decode=False*)
Return the current payload, which will be a list of *Message* objects when *is_multipart()* is `True`, or a string when *is_multipart()* is `False`. If the payload is a list and you mutate the list object, you modify the message's payload in place.

With optional argument *i*, *get_payload()* will return the *i*-th element of the payload, counting from zero, if *is_multipart()* is `True`. An *IndexError* will be raised if *i* is less than 0 or greater than or equal to the number of items in the payload. If the payload is a string (i.e. *is_multipart()* is `False`) and *i* is given, a *TypeError* is raised.

Optional *decode* is a flag indicating whether the payload should be decoded or not, according to the *Content-Transfer-Encoding* header. When `True` and the message is not a multipart, the payload will be decoded if this header's value is `quoted-printable` or `base64`. If some other encoding is used, or *Content-Transfer-Encoding* header is missing, the payload is returned as-is (undecoded). In all cases the returned value is binary data. If the message is a multipart and the *decode* flag is `True`, then `None` is returned. If the payload is base64 and it was not perfectly formed (missing padding, characters outside the base64 alphabet), then an appropriate defect will be added to the message's defect property (`InvalidBase64PaddingDefect` or `InvalidBase64CharactersDefect`, respectively).

When *decode* is `False` (the default) the body is returned as a string without decoding the *Content-Transfer-Encoding*. However, for a *Content-Transfer-Encoding* of 8bit, an attempt is made to decode the original bytes using the `charset` specified by the *Content-Type* header, using the `replace` error handler. If no `charset` is specified, or if the `charset` given is not recognized by the email package, the body is decoded using the default ASCII charset.

Esse é um método legado. Na `EmailMessage` classe sua funcionalidade é substituída por *get_content()* e *iter_parts()*.

set_payload(*payload*, *charset=None*)
Set the entire message object's payload to *payload*. It is the client's responsibility to ensure the payload invariants. Optional *charset* sets the message's default character set; see *set_charset()* for details.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by *set_content()*.

set_charset(*charset*)
Set the character set of the payload to *charset*, which can either be a *Charset* instance (see *email.charset*), a string naming a character set, or `None`. If it is a string, it will be converted to a *Charset* instance. If *charset* is `None`, the `charset` parameter will be removed from the *Content-Type* header (the message will not be otherwise modified). Anything else will generate a *TypeError*.

If there is no existing *MIME-Version* header one will be added. If there is no existing *Content-Type* header, one will be added with a value of *text/plain*. Whether the *Content-Type* header already exists or not, its `charset` parameter will be set to *charset.output_charset*. If *charset.input_charset* and *charset.output_charset* differ, the payload will be re-encoded to the *output_charset*. If there is no existing *Content-Transfer-Encoding* header, then the payload will be transfer-encoded, if needed, using the specified *Charset*, and a header with the appropriate value will be added. If a *Content-Transfer-Encoding* header already exists, the payload is assumed to already be correctly encoded using that *Content-Transfer-Encoding* and is not modified.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *charset* parameter of the `email.emailmessage.EmailMessage.set_content()` method.

get_charset()
Return the *Charset* instance associated with the message's payload.

This is a legacy method. On the EmailMessage class it always returns None.

The following methods implement a mapping-like interface for accessing the message's **RFC 2822** headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by *keys()*, but in a *Message* object, headers are always returned in the order they appeared in the original message, or were added to the message later. Any header deleted and then re-added are always appended to the end of the header list.

These semantic differences are intentional and are biased toward maximal convenience.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

In a model generated from bytes, any header values that (in contravention of the RFCs) contain non-ASCII bytes will, when retrieved through this interface, be represented as *Header* objects with a charset of *unknown-8bit*.

__len__()
Return the total number of headers, including duplicates.

__contains__(*name*)
Return True if the message object has a field named *name*. Matching is done case-insensitively and *name* should not include the trailing colon. Used for the in operator, e.g.:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)
Return the value of the named header field. *name* should not include the colon field separator. If the header is missing, None is returned; a *KeyError* is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the *get_all()* method to get the values of all the extant named headers.

__setitem__(*name*, *val*)
Add a header to the message with field name *name* and value *val*. The field is appended to the end of the message's existing fields.

Note that this does *not* overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name *name*, delete the field first, e.g.:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__(*name*)
Delete all occurrences of the field with name *name* from the message's headers. No exception is raised if the named field isn't present in the headers.

keys()
Return a list of all the message's header field names.

values()
Return a list of all the message's field values.

items()
Return a list of 2-tuples containing all the message's field headers and values.

get(*name*, *failobj=None*)
Return the value of the named header field. This is identical to *__getitem__()* except that optional *failobj* is returned if the named header is missing (defaults to `None`).

Here are some additional useful methods:

get_all(*name*, *failobj=None*)
Return a list of all the values for the field named *name*. If there are no such named headers in the message, *failobj* is returned (defaults to `None`).

add_header(*_name*, *_value*, ***_params*)
Extended header setting. This method is similar to *__setitem__()* except that additional header parameters can be provided as keyword arguments. *_name* is the header field to add and *_value* is the *primary* value for the header.

For each item in the keyword argument dictionary *_params*, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as `key="value"` unless the value is `None`, in which case only the key will be added. If the value contains non-ASCII characters, it can be specified as a three tuple in the format `(CHARSET, LANGUAGE, VALUE)`, where `CHARSET` is a string naming the charset to be used to encode the value, `LANGUAGE` can usually be set to `None` or the empty string (see **RFC 2231** for other possibilities), and `VALUE` is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII characters, it is automatically encoded in **RFC 2231** format using a `CHARSET` of `utf-8` and a `LANGUAGE` of `None`.

Here's an example:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

An example with non-ASCII characters:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

Que produz

```
Content-Disposition: attachment; filename*="iso-8859-1''Fu%DFballer.ppt"
```

replace_header(*_name*, *_value*)
Replace a header. Replace the first header found in the message that matches *_name*, retaining header order and field name case. If no matching header was found, a *KeyError* is raised.

get_content_type()
Return the message's content type. The returned string is coerced to lower case of the form *maintype/subtype*. If there was no *Content-Type* header in the message the default type as given by *get_default_type()* will be returned. Since according to **RFC 2045**, messages always have a default type, *get_content_type()* will always return a value.

RFC 2045 defines a message's default type to be *text/plain* unless it appears inside a *multipart/digest* container, in which case it would be *message/rfc822*. If the *Content-Type* header has an invalid type specification, **RFC 2045** mandates that the default type be *text/plain*.

get_content_maintype()
Return the message's main content type. This is the *maintype* part of the string returned by *get_content_type()*.

get_content_subtype()
Return the message's sub-content type. This is the *subtype* part of the string returned by *get_content_type()*.

get_default_type()
Return the default content type. Most messages have a default content type of *text/plain*, except for messages that are subparts of *multipart/digest* containers. Such subparts have a default content type of *message/rfc822*.

set_default_type(*ctype*)
Set the default content type. *ctype* should either be *text/plain* or *message/rfc822*, although this is not enforced. The default content type is not stored in the *Content-Type* header.

get_params(*failobj=None*, *header='content-type'*, *unquote=True*)
Return the message's *Content-Type* parameters, as a list. The elements of the returned list are 2-tuples of key/value pairs, as split on the `'='` sign. The left hand side of the `'='` is the key, while the right hand side is the value. If there is no `'='` sign in the parameter the value is the empty string, otherwise the value is as described in *get_param()* and is unquoted if optional *unquote* is `True` (the default).

Optional *failobj* is the object to return if there is no *Content-Type* header. Optional *header* is the header to search instead of *Content-Type*.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *params* property of the individual header objects returned by the header access methods.

get_param(*param*, *failobj=None*, *header='content-type'*, *unquote=True*)
Return the value of the *Content-Type* header's parameter *param* as a string. If the message has no *Content-Type* header or if there is no such parameter, then *failobj* is returned (defaults to `None`).

Optional *header* if given, specifies the message header to use instead of *Content-Type*.

Parameter keys are always compared case insensitively. The return value can either be a string, or a 3-tuple if the parameter was **RFC 2231** encoded. When it's a 3-tuple, the elements of the value are of the form `(CHARSET, LANGUAGE, VALUE)`. Note that both `CHARSET` and `LANGUAGE` can be `None`, in which case you should consider `VALUE` to be encoded in the `us-ascii` charset. You can usually ignore `LANGUAGE`.

If your application doesn't care whether the parameter was encoded as in **RFC 2231**, you can collapse the parameter value by calling *email.utils.collapse_rfc2231_value()*, passing in the return value from *get_param()*. This will return a suitably decoded Unicode string when the value is a tuple, or the original string unquoted if it isn't. For example:

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

In any case, the parameter value (either the returned string, or the `VALUE` item in the 3-tuple) is always unquoted, unless *unquote* is set to `False`.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *params* property of the individual header objects returned by the header access methods.

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)
Set a parameter in the *Content-Type* header. If the parameter already exists in the header, its value will be replaced with *value*. If the *Content-Type* header as not yet been defined for this message, it will be set to *text/plain* and the new parameter value will be appended as per **RFC 2045**.

Optional *header* specifies an alternative header to *Content-Type*, and all parameters will be quoted as necessary unless optional *requote* is `False` (the default is `True`).

If optional *charset* is specified, the parameter will be encoded according to **RFC 2231**. Optional *language* specifies the RFC 2231 language, defaulting to the empty string. Both *charset* and *language* should be strings.

If *replace* is `False` (the default) the header is moved to the end of the list of headers. If *replace* is `True`, the header will be updated in place.

Alterado na versão 3.4: Palavra-chave `replace` foi adicionada.

del_param(*param*, *header='content-type'*, *requote=True*)
Remove the given parameter completely from the *Content-Type* header. The header will be re-written in place without the parameter or its value. All values will be quoted as necessary unless *requote* is `False` (the default is `True`). Optional *header* specifies an alternative to *Content-Type*.

set_type(*type*, *header='Content-Type'*, *requote=True*)
Set the main type and subtype for the *Content-Type* header. *type* must be a string in the form *maintype/subtype*, otherwise a *ValueError* is raised.

This method replaces the *Content-Type* header, keeping all the parameters in place. If *requote* is `False`, this leaves the existing header's quoting as is, otherwise the parameters will be quoted (the default).

An alternative header can be specified in the *header* argument. When the *Content-Type* header is set a *MIME-Version* header is also added.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the `make_` and `add_` methods.

get_filename(*failobj=None*)
Return the value of the `filename` parameter of the *Content-Disposition* header of the message. If the header does not have a `filename` parameter, this method falls back to looking for the `name` parameter on the *Content-Type* header. If neither is found, or the header is missing, then *failobj* is returned. The returned string will always be unquoted as per *email.utils.unquote()*.

get_boundary(*failobj=None*)
Return the value of the `boundary` parameter of the *Content-Type* header of the message, or *failobj* if either the header is missing, or has no `boundary` parameter. The returned string will always be unquoted as per *email.utils.unquote()*.

set_boundary(*boundary*)
Set the `boundary` parameter of the *Content-Type* header to *boundary*. *set_boundary()* will always quote *boundary* if necessary. A *HeaderParseError* is raised if the message object has no *Content-Type* header.

Note that using this method is subtly different than deleting the old *Content-Type* header and adding a new one with the new boundary via *add_header()*, because *set_boundary()* preserves the order of the *Content-Type* header in the list of headers. However, it does *not* preserve any continuation lines which may have been present in the original *Content-Type* header.

get_content_charset(*failobj=None*)
Return the `charset` parameter of the *Content-Type* header, coerced to lower case. If there is no *Content-Type* header, or if that header has no `charset` parameter, *failobj* is returned.

Note that this method differs from *get_charset()* which returns the *Charset* instance for the default encoding of the message body.

get_charsets(*failobj=None*)
Return a list containing the character set names in the message. If the message is a *multipart*, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the `charset` parameter in the *Content-Type* header for the represented subpart. However, if the subpart has no *Content-Type* header, no `charset` parameter, or is not of the *text* main MIME type, then that item in the returned list will be *failobj*.

get_content_disposition()
Return the lowercased value (without parameters) of the message's *Content-Disposition* header if it has one, or None. The possible values for this method are *inline*, *attachment* or None if the message follows **RFC 2183**.

Novo na versão 3.5.

walk()
The *walk()* method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use *walk()* as the iterator in a for loop; each iteration returns the next subpart.

Here's an example that prints the MIME type of every part of a multipart message structure:

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

walk iterates over the subparts of any part where *is_multipart()* returns True, even though msg.get_content_maintype() == 'multipart' may return False. We can see this in our example by making use of the _structure debug helper function:

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

Here the message parts are not multiparts, but they do contain subparts. is_multipart() returns True and walk descends into the subparts.

Message objects can also optionally contain two instance attributes, which can be used when generating the plain text of a MIME message.

preamble
The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The *preamble* attribute contains this leading extra-armor text for MIME documents. When the `Parser` discovers some text after the headers but before the first boundary string, it assigns this text to the message's *preamble* attribute. When the `Generator` is writing out the plain text representation of a MIME message, and it finds the message has a *preamble* attribute, it will write this text in the area between the headers and the first boundary. See `email.parser` and `email.generator` for details.

Note that if the message object has no preamble, the *preamble* attribute will be `None`.

epilogue
The *epilogue* attribute acts the same way as the *preamble* attribute, except that it contains text that appears between the last boundary and the end of the message.

You do not need to set the epilogue to the empty string in order for the `Generator` to print a newline at the end of the file.

defects
The *defects* attribute contains a list of all the problems found when parsing this message. See `email.errors` for a detailed description of the possible parsing defects.

20.1.10 `email.mime`: Criando e-mail e objetos MIME fo zero

Código-fonte: Lib/email/mime/

Este módulo faz parte da API de e-mail legada (`Compat32`). Sua funcionalidade é parcialmente substituída por `contentmanager` na nova API, mas em certos aplicativos essas classes ainda podem ser úteis, mesmo em código não legado.

Normalmente, você obtém uma estrutura de objeto de mensagem passando um arquivo ou algum texto para um analisador, que analisa o texto e retorna o objeto de mensagem raiz. No entanto, você também pode criar uma estrutura de mensagem completa do zero, ou até objetos individuais de `Message` manualmente. De fato, você também pode pegar uma estrutura existente e adicionar novos objetos `Message`, movê-los, etc. Isso cria uma interface muito conveniente para fatiar e cortar dados de mensagens MIME.

Você pode criar uma nova estrutura de objeto criando instâncias de `Message`, adicionando anexos e todos os cabeçalhos apropriados manualmente. Porém, para mensagens MIME, o pacote `email` fornece algumas subclasses convenientes para facilitar as coisas.

Arquivo estão as classes:

class email.mime.base.**MIMEBase**(*_maintype*, *_subtype*, *, *policy=compat32*, ***_params*)
Módulo: `email.mime.base`

Esta é a classe base para todas as subclasses específicas de MIME de `Message`. Normalmente você não criará instâncias especificamente de `MIMEBase`, embora possa. A `MIMEBase` é fornecida principalmente como uma classe base conveniente para subclasses mais específicas para MIME.

_maintype é o tipo principal de `Content-Type` (ex., `text` ou `image`) e *_subtype* é o tipo principal de `Content-Type` (ex., `plain` ou `gif`). *_params* é um dicionário de parâmetros chave/valor e é passado diretamente para `Message.add_header`.

Se *policy* for especificado, (o padrão é a política `compat32`) será passado para `Message`.

A classe `MIMEBase` sempre adiciona um cabeçalho `Content-Type` (com base em *_maintype*, *_subtype* e *_params*) e um cabeçalho `MIME-Version` (sempre definido como `1.0`).

Alterado na versão 3.6: Adicionado parâmetro de apenas palavra reservada *policy*.

class email.mime.nonmultipart.**MIMENonMultipart**
Módulo: email.mime.nonmultipart

Uma subclasse de *MIMEBase*, esta é uma classe base intermediária para mensagens MIME que não são *multipart*. O principal objetivo desta classe é impedir o uso do método *attach()*, que só faz sentido para mensagens *multipart*. Se *attach()* for chamado, uma exceção *MultipartConversionError* será levantada.

class email.mime.multipart.**MIMEMultipart**(*_subtype='mixed'*, *boundary=None*, *_subparts=None*, ***, *policy=compat32*, ***_params*)
Módulo: email.mime.multipart

Uma subclasse de *MIMEBase*, esta é uma classe base intermediária para mensagens MIME que são *multipart*. O *_subtype * opcional é padronizado como *mixed*, mas pode ser usado para especificar o subtipo da mensagem. Um cabeçalho *Content-Type* de *multipart/_subtype* será adicionado ao objeto da mensagem. Um cabeçalho *MIME-Version* também será adicionado.

O *boundary* opcional é a string de limites de várias partes. Quando None (o padrão), o limite é calculado quando necessário (por exemplo, quando a mensagem é serializada).

_subparts é uma sequência de subpartes iniciais para a carga. Deve ser possível converter essa sequência em uma lista. Você sempre pode anexar novas subpartes à mensagem usando o método *Message.attach*.

O argumento opcional *policy* tem como padrão *compat32*.

Additional parameters for the *Content-Type* header are taken from the keyword arguments, or passed into the *_params* argument, which is a keyword dictionary.

Alterado na versão 3.6: Adicionado parâmetro de apenas palavra reservada *policy*.

class email.mime.application.**MIMEApplication**(*_data*, *_subtype='octet-stream'*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)
Módulo: email.mime.application

Uma subclasse de *MIMENonMultipart*, a classe *MIMEApplication* é usada para representar objetos de mensagem MIME do tipo principal *application*. *_data * é uma sequência que contém os dados brutos de bytes. O *_subtype* opcional especifica o subtipo MIME e o padrão é *octet-stream*.

O *_encoder* opcional é um chamável (isto é, função) que executará a codificação real dos dados para transporte. Esse chamável requer um argumento, que é a instância *MIMEApplication*. Ele deve usar *get_payload()* e *set_payload()* para alterar a carga útil para o formulário codificado. Também deve adicionar *Content-Transfer-Encoding* ou outros cabeçalhos ao objeto de mensagem, conforme necessário. A codificação padrão é base64. Veja o módulo *email.encoders* para obter uma lista dos codificadores embutidos.

O argumento opcional *policy* tem como padrão *compat32*.

_params são passados diretamente para o construtor da classe base.

Alterado na versão 3.6: Adicionado parâmetro de apenas palavra reservada *policy*.

class email.mime.audio.**MIMEAudio**(*_audiodata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)
Módulo: email.mime.audio

Uma subclasse de *MIMENonMultipart*, a classe *MIMEAudio* é usada para criar objetos de mensagem MIME do tipo principal *audio*. *_audiodata* é uma string que contém os dados de áudio não processados. Se esses dados puderem ser decodificados pelo módulo padrão do Python *sndhdr*, o subtipo será automaticamente incluído no cabeçalho *Content-Type*. Caso contrário, você pode especificar explicitamente o subtipo de áudio por meio do

argumento *_subtype*. Se o tipo menor não pôde ser adivinhado e *_subtype* não foi fornecido, então `TypeError` é levantado.

O *_encoder* opcional é um chamável (ou seja, função) que executará a codificação real dos dados de áudio para transporte. Esse chamável requer um argumento, que é a instância `MIMEAudio`. Ele deve usar : meth:*~email.message.Message.get_payload* e `set_payload()` para alterar a carga útil para a forma codificada. Também deve adicionar `Content-Transfer-Encoding` ou outros cabeçalhos ao objeto de mensagem, conforme necessário. A codificação padrão é base64. Veja o módulo `email.encoders` para obter uma lista dos codificadores embutidos.

O argumento opcional *policy* tem como padrão `compat32`.

_params são passados diretamente para o construtor da classe base.

Alterado na versão 3.6: Adicionado parâmetro de apenas palavra reservada *policy*.

class email.mime.image.**MIMEImage**(*_imagedata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

Módulo: `email.mime.image`

Uma subclasse de `MIMENonMultipart`, a classe `MIMEImage` é usada para criar objetos de mensagem MIME do tipo principal `image`. *_imagedata * é uma string que contém os dados brutos da imagem. Se esses dados puderem ser decodificados pelo módulo padrão do Python :mod:'imghdr', o subtipo será automaticamente incluído no cabeçalho :mailheader:'Content-Type'. Caso contrário, você pode especificar explicitamente o subtipo de imagem através do argumento *_subtype*.* Se o tipo menor não pôde ser adivinhado e *_subtype* não foi fornecido, então `TypeError` é levantado.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the image data for transport. This callable takes one argument, which is the `MIMEImage` instance. It should use `get_payload()` and `set_payload()` to change the payload to encoded form. It should also add any `Content-Transfer-Encoding` or other headers to the message object as necessary. The default encoding is base64. See the `email.encoders` module for a list of the built-in encoders.

O argumento opcional *policy* tem como padrão `compat32`.

_params are passed straight through to the `MIMEBase` constructor.

Alterado na versão 3.6: Adicionado parâmetro de apenas palavra reservada *policy*.

class email.mime.message.**MIMEMessage**(*_msg*, *_subtype='rfc822'*, ***, *policy=compat32*)

Module: `email.mime.message`

A subclass of `MIMENonMultipart`, the `MIMEMessage` class is used to create MIME objects of main type `message`. *_msg* is used as the payload, and must be an instance of class `Message` (or a subclass thereof), otherwise a `TypeError` is raised.

Optional *_subtype* sets the subtype of the message; it defaults to `rfc822`.

O argumento opcional *policy* tem como padrão `compat32`.

Alterado na versão 3.6: Adicionado parâmetro de apenas palavra reservada *policy*.

class email.mime.text.**MIMEText**(*_text*, *_subtype='plain'*, *_charset=None*, ***, *policy=compat32*)

```
`Module: :mod:\`email.mime.text\``
```

A subclass of `MIMENonMultipart`, the `MIMEText` class is used to create MIME objects of major type `text`. *_text* is the string for the payload. *_subtype* is the minor type and defaults to `plain`. *_charset* is the character set of the text and is passed as an argument to the `MIMENonMultipart` constructor; it defaults to `us-ascii` if the string contains only `ascii` code points, and `utf-8` otherwise. The *_charset* parameter accepts either a string or a `Charset` instance.

Unless the *_charset* argument is explicitly set to `None`, the MIMEText object created will have both a `Content-Type` header with a `charset` parameter, and a `Content-Transfer-Encoding` header. This means that a subsequent `set_payload` call will not result in an encoded payload, even if a charset is passed in the `set_payload` command. You can "reset" this behavior by deleting the `Content-Transfer-Encoding` header, after which a `set_payload` call will automatically encode the new payload (and add a new `Content-Transfer-Encoding` header).

O argumento opcional *policy* tem como padrão `compat32`.

Alterado na versão 3.5: *_charset* also accepts `Charset` instances.

Alterado na versão 3.6: Adicionado parâmetro de apenas palavra reservada *policy*.

20.1.11 `email.header`: Internationalized headers

Código Fonte: Lib/email/header.py

This module is part of the legacy (`Compat32`) email API. In the current API encoding and decoding of headers is handled transparently by the dictionary-like API of the `EmailMessage` class. In addition to uses in legacy code, this module can be useful in applications that need to completely control the character sets used when encoding headers.

The remaining text in this section is the original documentation of the module.

RFC 2822 is the base standard that describes the format of email messages. It derives from the older **RFC 822** standard which came into widespread use at a time when most email was composed of ASCII characters only. **RFC 2822** is a specification written assuming email contains only 7-bit ASCII characters.

Of course, as email has been deployed worldwide, it has become internationalized, such that language specific character sets can now be used in email messages. The base standard still requires email messages to be transferred using only 7-bit ASCII characters, so a slew of RFCs have been written describing how to encode email containing non-ASCII characters into **RFC 2822**-compliant format. These RFCs include **RFC 2045**, **RFC 2046**, **RFC 2047**, and **RFC 2231**. The `email` package supports these standards in its `email.header` and `email.charset` modules.

If you want to include non-ASCII characters in your email headers, say in the `Subject` or `To` fields, you should use the `Header` class and assign the field in the `Message` object to an instance of `Header` instead of using a string for the header value. Import the `Header` class from the `email.header` module. For example:

```
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> msg.as_string()
'Subject: =?iso-8859-1?q?p=F6stal?=\n\n'
```

Notice here how we wanted the `Subject` field to contain a non-ASCII character? We did this by creating a `Header` instance and passing in the character set that the byte string was encoded in. When the subsequent `Message` instance was flattened, the `Subject` field was properly **RFC 2047** encoded. MIME-aware mail readers would show this header using the embedded ISO-8859-1 character.

Here is the `Header` class description:

class `email.header.`**Header**(*s=None*, *charset=None*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*, *errors='strict'*)

Create a MIME-compliant header that can contain strings in different character sets.

Optional *s* is the initial header value. If `None` (the default), the initial header value is not set. You can later append to the header with `append()` method calls. *s* may be an instance of `bytes` or `str`, but see the `append()` documentation for semantics.

Optional *charset* serves two purposes: it has the same meaning as the *charset* argument to the `append()` method. It also sets the default character set for all subsequent `append()` calls that omit the *charset* argument. If *charset* is not provided in the constructor (the default), the `us-ascii` character set is used both as *s*'s initial charset and as the default for subsequent `append()` calls.

The maximum line length can be specified explicitly via *maxlinelen*. For splitting the first line to a shorter value (to account for the field header which isn't included in *s*, e.g. `Subject`) pass in the name of the field in *header_name*. The default *maxlinelen* is 76, and the default value for *header_name* is `None`, meaning it is not taken into account for the first line of a long, split header.

Optional *continuation_ws* must be **RFC 2822**-compliant folding whitespace, and is usually either a space or a hard tab character. This character will be prepended to continuation lines. *continuation_ws* defaults to a single space character.

Optional *errors* is passed straight through to the `append()` method.

append(*s*, *charset=None*, *errors='strict'*)
Append the string *s* to the MIME header.

Optional *charset*, if given, should be a `Charset` instance (see `email.charset`) or the name of a character set, which will be converted to a `Charset` instance. A value of `None` (the default) means that the *charset* given in the constructor is used.

s may be an instance of `bytes` or `str`. If it is an instance of `bytes`, then *charset* is the encoding of that byte string, and a `UnicodeError` will be raised if the string cannot be decoded with that character set.

If *s* is an instance of `str`, then *charset* is a hint specifying the character set of the characters in the string.

In either case, when producing an **RFC 2822**-compliant header using **RFC 2047** rules, the string will be encoded using the output codec of the charset. If the string cannot be encoded using the output codec, a UnicodeError will be raised.

Optional *errors* is passed as the errors argument to the decode call if *s* is a byte string.

encode(*splitchars=';, \t'*, *maxlinelen=None*, *linesep='\n'*)
Encode a message header into an RFC-compliant format, possibly wrapping long lines and encapsulating non-ASCII parts in base64 or quoted-printable encodings.

Optional *splitchars* is a string containing characters which should be given extra weight by the splitting algorithm during normal header wrapping. This is in very rough support of **RFC 2822**'s 'higher level syntactic breaks': split points preceded by a splitchar are preferred during line splitting, with the characters preferred in the order in which they appear in the string. Space and tab may be included in the string to indicate whether preference should be given to one over the other as a split point when other split chars do not appear in the line being split. Splitchars does not affect **RFC 2047** encoded lines.

maxlinelen, if given, overrides the instance's value for the maximum line length.

linesep specifies the characters used to separate the lines of the folded header. It defaults to the most useful value for Python application code (`\n`), but `\r\n` can be specified in order to produce headers with RFC-compliant line separators.

Alterado na versão 3.2: Adicionado o argumento *linesep*.

The `Header` class also provides a number of methods to support standard operators and built-in functions.

__str__()
Returns an approximation of the `Header` as a string, using an unlimited line length. All pieces are converted

to unicode using the specified encoding and joined together appropriately. Any pieces with a charset of `'unknown-8bit'` are decoded as ASCII using the `'replace'` error handler.

Alterado na versão 3.2: Added handling for the `'unknown-8bit'` charset.

__eq__(*other*)
This method allows you to compare two *Header* instances for equality.

__ne__(*other*)
This method allows you to compare two *Header* instances for inequality.

The *email.header* module also provides the following convenient functions.

email.header.**decode_header**(*header*)
Decode a message header value without converting the character set. The header value is in *header*.

This function returns a list of `(decoded_string, charset)` pairs containing each of the decoded parts of the header. *charset* is `None` for non-encoded parts of the header, otherwise a lower case string containing the name of the character set specified in the encoded string.

Here's an example:

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[(b'p\xf6stal', 'iso-8859-1')]
```

email.header.**make_header**(*decoded_seq*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*)
Create a *Header* instance from a sequence of pairs as returned by *decode_header()*.

decode_header() takes a header value string and returns a sequence of pairs of the format `(decoded_string, charset)` where *charset* is the name of the character set.

This function takes one of those sequence of pairs and returns a *Header* instance. Optional *maxlinelen*, *header_name*, and *continuation_ws* are as in the *Header* constructor.

20.1.12 email.charset: Representing character sets

Código Fonte: Lib/email/charset.py

This module is part of the legacy (`Compat32`) email API. In the new API only the aliases table is used.

The remaining text in this section is the original documentation of the module.

This module provides a class *Charset* for representing character sets and character set conversions in email messages, as well as a character set registry and several convenience methods for manipulating this registry. Instances of *Charset* are used in several other modules within the *email* package.

Import this class from the *email.charset* module.

class email.charset.**Charset**(*input_charset=DEFAULT_CHARSET*)
Map character sets to their email properties.

This class provides information about the requirements imposed on email for a specific character set. It also provides convenience routines for converting between character sets, given the availability of the applicable codecs. Given a character set, it will do its best to provide information on how to use that character set in an email message in an RFC-compliant way.

Certain character sets must be encoded with quoted-printable or base64 when used in email headers or bodies. Certain character sets must be converted outright, and are not allowed in email.

Optional *input_charset* is as described below; it is always coerced to lower case. After being alias normalized it is also used as a lookup into the registry of character sets to find out the header encoding, body encoding, and output conversion codec to be used for the character set. For example, if *input_charset* is `iso-8859-1`, then headers and bodies will be encoded using quoted-printable and no output conversion codec is necessary. If *input_charset* is `euc-jp`, then headers will be encoded with base64, bodies will not be encoded, but output text will be converted from the `euc-jp` character set to the `iso-2022-jp` character set.

Charset instances have the following data attributes:

input_charset
The initial character set specified. Common aliases are converted to their *official* email names (e.g. `latin_1` is converted to `iso-8859-1`). Defaults to 7-bit `us-ascii`.

header_encoding
If the character set must be encoded before it can be used in an email header, this attribute will be set to `Charset.QP` (for quoted-printable), `Charset.BASE64` (for base64 encoding), or `Charset.SHORTEST` for the shortest of QP or BASE64 encoding. Otherwise, it will be `None`.

body_encoding
Same as *header_encoding*, but describes the encoding for the mail message's body, which indeed may be different than the header encoding. `Charset.SHORTEST` is not allowed for *body_encoding*.

output_charset
Some character sets must be converted before they can be used in email headers or bodies. If the *input_charset* is one of them, this attribute will contain the name of the character set output will be converted to. Otherwise, it will be `None`.

input_codec
The name of the Python codec used to convert the *input_charset* to Unicode. If no conversion codec is necessary, this attribute will be `None`.

output_codec
The name of the Python codec used to convert Unicode to the *output_charset*. If no conversion codec is necessary, this attribute will have the same value as the *input_codec*.

Charset instances also have the following methods:

get_body_encoding()
Return the content transfer encoding used for body encoding.

This is either the string `quoted-printable` or `base64` depending on the encoding used, or it is a function, in which case you should call the function with a single argument, the Message object being encoded. The function should then set the *Content-Transfer-Encoding* header itself to whatever is appropriate.

Returns the string `quoted-printable` if *body_encoding* is `QP`, returns the string `base64` if *body_encoding* is `BASE64`, and returns the string `7bit` otherwise.

get_output_charset()
Return the output character set.

This is the *output_charset* attribute if that is not `None`, otherwise it is *input_charset*.

header_encode(*string*)
Header-encode the string *string*.

The type of encoding (base64 or quoted-printable) will be based on the *header_encoding* attribute.

header_encode_lines(*string*, *maxlengths*)
Header-encode a *string* by converting it first to bytes.

This is similar to *header_encode()* except that the string is fit into maximum line lengths as given by the argument *maxlengths*, which must be an iterator: each element returned from this iterator will provide the next maximum line length.

body_encode(*string*)
Body-encode the string *string*.

The type of encoding (base64 or quoted-printable) will be based on the *body_encoding* attribute.

The *Charset* class also provides a number of methods to support standard operations and built-in functions.

__str__()
Returns *input_charset* as a string coerced to lower case. `__repr__()` is an alias for *__str__()*.

__eq__(*other*)
This method allows you to compare two *Charset* instances for equality.

__ne__(*other*)
This method allows you to compare two *Charset* instances for inequality.

The *email.charset* module also provides the following functions for adding new entries to the global character set, alias, and codec registries:

email.charset.**add_charset**(*charset*, *header_enc=None*, *body_enc=None*, *output_charset=None*)
Add character properties to the global registry.

charset is the input character set, and must be the canonical name of a character set.

Optional *header_enc* and *body_enc* is either `Charset.QP` for quoted-printable, `Charset.BASE64` for base64 encoding, `Charset.SHORTEST` for the shortest of quoted-printable or base64 encoding, or `None` for no encoding. `SHORTEST` is only valid for *header_enc*. The default is `None` for no encoding.

Optional *output_charset* is the character set that the output should be in. Conversions will proceed from input charset, to Unicode, to the output charset when the method `Charset.convert()` is called. The default is to output in the same character set as the input.

Both *input_charset* and *output_charset* must have Unicode codec entries in the module's character set-to-codec mapping; use *add_codec()* to add codecs the module does not know about. See the *codecs* module's documentation for more information.

The global character set registry is kept in the module global dictionary `CHARSETS`.

email.charset.**add_alias**(*alias*, *canonical*)
Add a character set alias. *alias* is the alias name, e.g. `latin-1`. *canonical* is the character set's canonical name, e.g. `iso-8859-1`.

The global charset alias registry is kept in the module global dictionary `ALIASES`.

email.charset.**add_codec**(*charset*, *codecname*)
Add a codec that map characters in the given character set to and from Unicode.

charset is the canonical name of a character set. *codecname* is the name of a Python codec, as appropriate for the second argument to the *str*'s *encode()* method.

20.1.13 email.encoders: Encoders

Código Fonte: Lib/email/encoders.py

This module is part of the legacy (Compat32) email API. In the new API the functionality is provided by the *cte* parameter of the *set_content()* method.

This module is deprecated in Python 3. The functions provided here should not be called explicitly since the *MIMEText* class sets the content type and CTE header using the *_subtype* and *_charset* values passed during the instaniation of that class.

The remaining text in this section is the original documentation of the module.

When creating *Message* objects from scratch, you often need to encode the payloads for transport through compliant mail servers. This is especially true for *image/** and *text/** type messages containing binary data.

The *email* package provides some convenient encodings in its encoders module. These encoders are actually used by the *MIMEAudio* and *MIMEImage* class constructors to provide default encodings. All encoder functions take exactly one argument, the message object to encode. They usually extract the payload, encode it, and reset the payload to this newly encoded value. They should also set the *Content-Transfer-Encoding* header as appropriate.

Note that these functions are not meaningful for a multipart message. They must be applied to individual subparts instead, and will raise a *TypeError* if passed a message whose type is multipart.

Here are the encoding functions provided:

email.encoders.**encode_quopri**(*msg*)
: Encodes the payload into quoted-printable form and sets the *Content-Transfer-Encoding* header to quoted-printable[1]. This is a good encoding to use when most of your payload is normal printable data, but contains a few unprintable characters.

email.encoders.**encode_base64**(*msg*)
: Encodes the payload into base64 form and sets the *Content-Transfer-Encoding* header to base64. This is a good encoding to use when most of your payload is unprintable data since it is a more compact form than quoted-printable. The drawback of base64 encoding is that it renders the text non-human readable.

email.encoders.**encode_7or8bit**(*msg*)
: This doesn't actually modify the message's payload, but it does set the *Content-Transfer-Encoding* header to either 7bit or 8bit as appropriate, based on the payload data.

email.encoders.**encode_noop**(*msg*)
: This does nothing; it doesn't even set the *Content-Transfer-Encoding* header.

20.1.14 email.utils: Utilitários diversos

Código Fonte: Lib/email/utils.py

Existem alguns utilitários úteis fornecidos no *email.utils* module:

email.utils.**localtime**(*dt=None*)
: Retorna a hora local como um objeto datetime com reconhecimento. Se chamado sem argumentos, retorne a hora atual. Caso contrário, o argumento *dt* deve ser uma instância: 'class': *~datetime.datetime* e é convertido para o fuso horário local de acordo com o banco de dados de fuso horário do sistema. Se *dt* é ingênuo (isto é, dt.tzinfo é None), presume-se que seja na hora local. Neste caso, um valor positivo ou zero para * isdst * faz com que " localtime" presuma inicialmente que o horário de verão (por exemplo, horário de verão) esteja ou não

[1] Note that encoding with *encode_quopri()* also encodes all tabs and space characters in the data.

(respectivamente) em vigor pelo tempo especificado. Um valor negativo para *isdst* faz com que o `localtime` tente adivinhar se o horário de verão está em vigor pelo tempo especificado.

Novo na versão 3.3.

`email.utils.`**`make_msgid`**(*idstring=None*, *domain=None*)
Retorna uma string adequada para um cabeçalho: rfc: *2822* -compliant: mailheader:‘ Message-ID‘. O opcional * idstring *, se fornecido, é uma string usada para fortalecer a exclusividade do ID da mensagem. Opcional * domínio * se dado fornece a porção do msgid após o ‘@’. O padrão é o nome do host local. Normalmente, não é necessário substituir esse padrão, mas pode ser útil em alguns casos, como um sistema distribuído de construção que usa um nome de domínio consistente em vários hosts.

Alterado na versão 3.2: Adicionado a palavra-chave *domain*.

As funções restantes fazem parte da API de e-mail herdada (`Compat32`). Não há necessidade de usá-las diretamente com a nova API, pois a análise e a formatação fornecidas são feitas automaticamente pelo mecanismo de análise de cabeçalhos da nova API.

`email.utils.`**`quote`**(*str*)
Devolve uma nova string com barras invertidas em *str* substituídas por duas barras invertidas e aspas duplas substituídas por aspas duplas invertidas.

`email.utils.`**`unquote`**(*str*)
Retorna uma nova string que é uma *versão* sem nome *str*. Se *str* terminar e começar com aspas duplas, elas serão removidas. Da mesma forma, se *str* termina e começa com colchetes angulares, eles são removidos.

`email.utils.`**`parseaddr`**(*address*)
Analisar endereço – que deve ser o valor de algum campo contendo endereço, como: mailheader: *To* ou: mailheader:*Cc* - em suas partes constituent *realname * e *email address*. Retorna uma tupla daquela informação, a menos que a análise falhe, caso em que uma 2-tupla de `('', '')` é retornada.

`email.utils.`**`formataddr`**(*pair*, *charset='utf-8'*)
O inverso de: meth: *parseaddr*, isto leva uma 2-tupla do formulário `(realname, email_address)` e retorna o valor de string adequado para um cabeçalho: mailheader:*To* ou: mailheader: *Cc* . Se o primeiro elemento de *pair* for falso, o segundo elemento será retornado sem modificações.

Opcional *charset* é o conjunto de caracteres que será usado na codificação **RFC 2047** do‘ *realname* ‘ se o“realname“ contiver caracteres não-ASCII. Pode ser uma instância de: classe: *str* ou a: class:‘ ~ email.charset.Charset‘. O padrão é `utf-8`.

Alterado na versão 3.3: Adicionado a opção *charset*.

`email.utils.`**`getaddresses`**(*fieldvalues*)
Este método retorna uma lista de 2 tuplas do formulário retornado por `parseaddr ()`. * fieldvalues * é uma sequência de valores do campo de cabeçalho que pode ser retornada por:meth: *Message.get_all* <*email.message.Message.get_all*>. Aqui está um exemplo simples que recebe todos os destinatários de uma message:

```
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

`email.utils.`**`parsedate`**(*date*)
Tenta analisar uma data de acordo com as regras em **RFC 2822**. no entanto, alguns mailers não seguem esse formato conforme especificado, portanto *`parsedate()`* tenta adivinhar corretamente em tais casos. *date* é uma string contendo uma data **RFC 2822**, como“” Mon, 20 Nov 1995 19:12:08 -0500 “. `Se`

conseguir analisar a data, :func:`parsedate` retorna uma 9-tupla que pode ser passada diretamente para :func:`time.mktime`; caso contrário, ``None será retornado. Observe que os índices 6, 7 e 8 da tupla de resultados não são utilizáveis.

email.utils.**parsedate_tz**(*date*)
Executa a mesma função que *parsedate()*, mas retorna None ou uma tupla de 10; os primeiros 9 elementos formam uma tupla que pode ser passada diretamente para *time.mktime()*, e o décimo é o deslocamento do fuso horário da data do UTC (que é o termo oficial para o horário de Greenwich)[1]. Se a string de entrada não tem fuso horário, o último elemento da tupla retornado é 0, que representa UTC. Observe que os índices 6, 7 e 8 da tupla de resultado não podem ser usados.

email.utils.**parsedate_to_datetime**(*date*)
O inverso de *format_datetime()*. Desempenha a mesma função que *parsedate()*, mas em caso de sucesso retorna um *datetime*. Se a data de entrada tem um fuso horário de -0000, o datetime será um datetime ingênuo, e se a data estiver em conformidade com os RFCs representará um horário em UTC, mas sem indicação do fuso horário de origem real da mensagem de onde vem a data. Se a data de entrada tiver qualquer outro deslocamento de fuso horário válido, o datetime será um datetime consciente com o correspondente a *timezone tzinfo*.

Novo na versão 3.3.

email.utils.**mktime_tz**(*tuple*)
Transforma uma tupla de 10 conforme retornado por *parsedate_tz()* em um timestamp UTC (segundos desde a Era Unix). Se o item de fuso horário na tupla for None, considera a hora local.

email.utils.**formatdate**(*timeval=None*, *localtime=False*, *usegmt=False*)
Retorna uma string de data conforme **RFC 2822**. Por exemplo:

```
Fri, 09 Nov 2001 01:08:47 -0000
```

O *timeval* opcional, se fornecido, é um valor de tempo de ponto flutuante, conforme aceito por *time.gmtime()* e *time.localtime()*, caso contrário, o tempo atual é usado.

Há um sinalizador opcional *localtime*, que, quando é True, interpreta *timeval* e retorna uma data relativa ao fuso horário local em vez do UTC, levando em consideração o horário de verão. O padrão é False, o que significa que o UTC é usado.

O *usegmt* opcional é um sinalizador que quando True, produz uma string de data com o fuso horário como uma string ascii GMT, ao invés de um numérico -0000. Isso é necessário para alguns protocolos (como HTTP). Isso se aplica apenas quando *localtime* for False. O padrão é False.

email.utils.**format_datetime**(*dt*, *usegmt=False*)
Como formatdate, mas a entrada é uma instância de *datetime*. Se for uma data e hora ingênua, será considerado "UTC sem informações sobre o fuso horário de origem" e o convencional -0000 será usado para o fuso horário. Se for um datetime ciente, então o deslocamento de fuso horário numérico é usado. Se for um fuso horário ciente com deslocamento zero, então *usegmt* pode ser definido como True, caso em que a string GMT é usada em vez do deslocamento numérico do fuso horário. Isso fornece uma maneira de gerar cabeçalhos de data HTTP em conformidade com os padrões.

Novo na versão 3.3.

email.utils.**decode_rfc2231**(*s*)
Decodifica a string *s* de acordo com **RFC 2231**.

email.utils.**encode_rfc2231**(*s*, *charset=None*, *language=None*)
Codifica a string *s* de acordo com **RFC 2231**. *charset* e *language* opcionais, se fornecido, são o nome do conjunto de caracteres e o nome do idioma a ser usado. Se nenhum deles for fornecido, *s* é retornado como está. Se *charset* for fornecido, mas *language* não, a string será codificada usando a string vazia para *language*.

[1] Observa que o sinal do deslocamento de fuso horário é o oposto do sinal da variável time.timezone para o mesmo fuso horário; a última variável segue o padrão POSIX enquanto este módulo segue **RFC 2822**.

email.utils.**collapse_rfc2231_value**(*value*, *errors='replace'*, *fallback_charset='us-ascii'*)
Quando um parâmetro de cabeçalho é codificado no formato **RFC 2231**, `Message.get_param` pode retornar uma tupla de 3 contendo o conjunto de caracteres, idioma e valor. `collapse_rfc2231_value()` transforma isso em uma string Unicode. *errors* opcionais são passados para o argumento *errors* do método `encode()` de `str`; o padrão é `'replace'`. *fallback_charset* opcional especifica o conjunto de caracteres a ser usado se aquele no cabeçalho **RFC 2231** não for conhecido pelo Python; o padrão é `'us-ascii'`.

Por conveniência, se *value* passado para `collapse_rfc2231_value()` não for uma tupla, deve ser uma string e é retornado sem aspas.

email.utils.**decode_params**(*params*)
Decodifica a lista de parâmetros de acordo com **RFC 2231**. *params* é uma sequência de 2 tuplas contendo elementos do formulário `(content-type, string-value)`.

20.1.15 email.iterators: Iteradores

Código Fonte: Lib/email/iterators.py

A iteração sobre uma árvore de objetos de mensagem é bastante fácil com o método `Message.walk`. O módulo `email.iterators` fornece algumas iterações úteis de nível superior sobre as árvores de objetos de mensagens.

email.iterators.**body_line_iterator**(*msg*, *decode=False*)
Isso itera sobre todas as cargas úteis em todas as subpartes de *msg*, retornando as cargas úteis das strings de linhas por linha. Ele pula todos os cabeçalhos da subparte e pula qualquer subparte com uma carga útil que não seja uma string Python. Isso é um pouco equivalente à leitura da representação de texto simples da mensagem de um arquivo usando `readline()`, pulando todos os cabeçalhos intermediários.

decode opcional é passado por meio do `Message.get_payload`.

email.iterators.**typed_subpart_iterator**(*msg*, *maintype='text'*, *subtype=None*)
Isso itera sobre todas as subpartes de *msg*, retornando apenas as subpartes que correspondem ao tipo MIME especificado por *maintype* e *subtype*.

Observe que *subtipo* é opcional; se omitido, a correspondência de tipo MIME da subparte é feita apenas com o tipo principal. *maintype* também é opcional; o padrão é `text`.

Assim, por padrão `typed_subpart_iterator()` retorna cada subparte que possui um tipo MIME de `text/*`.

A seguinte função foi adicionada como uma ferramenta de depuração útil. *Não* deve ser considerado parte da interface pública suportada para o pacote.

email.iterators.**_structure**(*msg*, *fp=None*, *level=0*, *include_default=False*)
Imprime uma representação recuada dos tipos de conteúdo da estrutura do objeto de mensagem. Por exemplo:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
    text/plain
    text/plain
    multipart/digest
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
```

(continua na próxima página)

(continuação da página anterior)

```
        message/rfc822
            text/plain
        message/rfc822
            text/plain
    text/plain
```

O *fp* opcional é um objeto arquivo ou similar para o qual deve-se imprimir a saída. Deve ser adequado para a função Python `print()`. *level usado internamente. *include_default*, se verdadeiro, também imprime o tipo padrão.

Ver também:

Módulo *smtplib* Cliente SMTP (Simple Mail Transport Protocol)

Módulo *poplib* Cliente POP (Post Office Protocol)

Módulo *imaplib* Cliente IMAP (Internet Messsge Access Protocol)

Módulo *nntplib* Cliente NNTP (Network News Transport Protocol)

Módulo *mailbox* Ferramentas para criar, ler, e gerenciar coleções de mensagem em disco usando vários formatos padrão.

Módulo *smtpd* Framework de servidor SMTP (primeiramente usado para testes)

20.2 `json` — JSON codificador e decodificador

Código Fonte: Lib/json/__init__.py

JSON (JavaScript Object Notation), especificado pelo **RFC 7159** (que tornou a **RFC 4627** obsoleta) e pelo ECMA-404, é um formato leve de troca de dados inspirado pelo sintaxe de objeto JavaScript (embora não seja um subconjunto estrito de JavaScript[1]).

Aviso: Be cautious when parsing JSON data from untrusted sources. A malicious JSON string may cause the decoder to consume considerable CPU and memory resources. Limiting the size of data to be parsed is recommended.

json expõe uma API familiar para pessoas usuárias dos módulos *marshal* e *pickle* da biblioteca padrão.

Codificação de hierarquias básicas de objetos Python:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
```

(continua na próxima página)

[1] Como apresentado na errata para RFC 7159, JSON permite os caracteres literais U+2028 (SEPARADOR DE LINHA) e U+2029 (SEPARADOR DE PARÁGRAFO) em strings, enquanto que JavaScript (ECMAScript Edition 5.1) não.

(continuação da página anterior)

```
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

Codificação compacta:

```
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

Saída bonita:

```
>>> import json
>>> print(json.dumps({'4': 5, '6': 7}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

Decodificando JSON:

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
['streaming API']
```

Especialização em decodificação de objeto JSON:

```
>>> import json
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')
```

Estendendo *JSONEncoder*:

```
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise the TypeError
...         return json.JSONEncoder.default(self, obj)
```

(continua na próxima página)

(continuação da página anterior)

```
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[2.0', ', 1.0', ']']
```

Usando *json.tool* para validar a partir do console e exibir formatado:

```
$ echo '{"json":"obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

Veja *Interface de Linha de Comando* para a documentação detalhada.

Nota: JSON é um subconjunto da YAML 1.2. O JSON gerado pelas definições padrão desse módulo (particularmente, o valor padrão dos *separadores*) é também um subconjunto da YAML 1.0 e 1.1. Esse módulo pode, por tanto, também ser usado para seriar YAML.

20.2.1 Uso Básico

`json.`**`dump`**(*obj*, *fp*, ***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Serializa um *obj* formatado como um stream JSON para *fp* (um `.write()`- dando suporte a objeto arquivo ou similar) usando essa *tabela de conversão*.

Se *skipkeys* for verdadeiro (default: `False`), as chaves de dicionário que não forem de um tipo básico (*str*, *int*, *float*, *bool*, `None`) serão ignoradas ao invés de levantarem um *TypeError*.

O módulo *json* sempre gera objetos *str*, e não objetos *bytes* . Dessa forma, `fp.write()` precisa suportar entradas *str*.

If *ensure_ascii* is true (the default), the output is guaranteed to have all incoming non-ASCII characters escaped. If *ensure_ascii* is false, these characters will be output as-is.

Se *check_circular* for falso (default: `True`), então a checagem referência circular para tipos containers será ignorada e uma referência circular resultará em um *OverflowError* (ou pior).

Se *allow_nan* for falso (padrão: `True`), serializar valores *float* fora do intervalo (`nan`, `inf`, `-inf`) em estrita conformidade com a especificação JSON será um *ValueError* . Se *allow_nan* for verdadeiro, seus equivalentes JavaScript (`NaN`, `Infinity`, `-Infinity`) serão usados.

Se *indent* for um inteiro não-negativo ou uma string, então elementos de um vetor JSON e membros de objetos terão uma saída formatada com este nível de identação. Um n;ivel de identação 0, negativo ou `""` apenas colocará novas linhas. `None` (o padrão) seleciona a presentação mais compacta. Usando um inteiro positivo a identação terá alguns espaços por nível. Se *indent* for uma string (como `"\t"`), essa string será usada para identar cada nível.

Alterado na versão 3.2: Permite strings para *indent*, além de inteiros.

Se especificado, *separators* deve ser uma tupla `(item_separator, key_separator)`. O padrão é `(', ', ': ')` se *indent* for `None` e `(',', ': ')` caso contrário. Para pegar representação JSON mais compacta, você deve especificar `(',', ':')` para eliminar espaços em branco.

Alterado na versão 3.4: Usa `(', ', ': ')` como padrão se *indent* não for `None`.

Se especificado, *default* deve ser uma função para ser chamada para objetos que não podem ser serializados de outra forma. Deve retornar uma versão codificada JSON do objeto ou lançar uma exceção `TypeError`. Se não for especificada, `TypeError` é levantada.

Se *sort_keys* for verdadeiro (padrão: `False`), então os dicionários da saída serão ordenados pela chave.

Para usar uma subclasse de `JSONEncoder` personalizada (por ex., uma que sobrescreve o método `default()` para serializar tipos adicionais), especifique isso com argumento *cls*; caso contrário é usado `JSONEncoder`.

Alterado na versão 3.6: Todos os parâmetros opcionais agora são *somente-nomeado*.

Nota: Diferente de `pickle` e `marshal`, JSON não é um protocolo com datagrama, assim tentar serializar múltiplos objetos com chamadas repetidas para `dump()` usando o mesmo *fp* resultará em um arquivo JSON inválido.

`json.`**`dumps`**(*obj*, *, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Serializa *obj* para uma `str` com formato JSON usando essa *tabela de conversão*. Os argumentos têm o mesmo significado que na função `dump()`.

Nota: Chaves nos pares chave/valor de JSON são sempre do tipo `str`. Quando um dicionário é convertido para JSON, todas as chaves são convertidas para strings. Como resultado disso, se um dicionário é convertido para JSON e depois de volta para o dicionários, o dicionário pode não ser igual ao original. Isto é, `loads(dumps(x)) != x` se x tem chaves não-strings.

`json.`**`load`**(*fp*, *, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

Desserializa *fp* (um *arquivo texto* ou *arquivo binário* com suporte a `.read()` contendo um documento JSON) para um objeto Python usando essa *tabela de conversão*.

object_hook é uma função opcional que será chamada com o resultado de qualquer objeto literal decodificado (um `dict`). O valor do retorno de *object_hook* será usado no lugar de `dict`. Esse recurso pode ser usado para implementar decodificadores personalizados (por exemplo, sugestão para classes JSON-RPC).

object_pairs_hook é uma função opcional que será chamada com o resultado de qualquer objeto literal decodificado com uma lista ordenada de pares. O valor de retorno de *object_pairs_hook* será usado no lugar do `dict`. Este recurso pode ser usado para implementar decodificadores personalizados. Se *object_pairs_hook* também for definido, o *object_pairs_hook* terá prioridade.

Alterado na versão 3.1: Adicionado suporte para *object_pairs_hook*.

parse_float, se especificado, será chamada com a string de cada ponto flutuante JSON para ser decodificado. Por padrão, é equivalente a `float(num_str)`. Pode ser usado para qualquer outro tipo de dado ou conversor para ponto flutuante JSON (ex. `decimal.Decimal`).

parse_int, se especificado, será chamada com a string de cada inteiro JSON para ser decodificado. Por padrão, é equivalente a `int(num_str)`. Pode ser usado para qualquer outro tipo de dado ou conversor para inteiro JSON (ex. `float`).

Alterado na versão 3.7.14: The default *parse_int* of `int()` now limits the maximum length of the integer string via the interpreter's *integer string conversion length limitation* to help avoid denial of service attacks.

parse_constant, se especificado, será chamada para cada um das seguintes strings: `'-Infinity'`, `'Infinity'`, `'NaN'`. Isso pode ser usado para levantar uma exceção se forem encontrados números JSON inválidos.

Alterado na versão 3.1: *parse_constant* não é mais chamada para 'null', 'true', 'false'.

Para usar uma subclasse de `JSONDecoder` personalizada, especifique isto com o argumento nomeado `cls`; caso contrário será usada `JSONDecoder`. Argumentos nomeados adicionais poderão ser passados para o construtor da classe.

Se os dados a serem desserializados não forem um documento JSON válido, será levantada uma exceção `JSONDecodeError`.

Alterado na versão 3.6: Todos os parâmetros opcionais agora são *somente-nomeado*.

Alterado na versão 3.6: *fp* agora pode ser um *arquivo binário*. A entrada deve estar codificada como UTF-8, UTF-16 ou UTF-32.

`json.`**`loads`**(*s*, *, *encoding=None*, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

Desserializa *s* (uma instancia de `str`, `bytes` ou `bytearray` contendo um documento JSON) para um objeto Python essa *tabela de conversão*.

The other arguments have the same meaning as in `load()`, except *encoding* which is ignored and deprecated.

Se os dados a serem desserializados não forem um documento JSON válido, será levantada uma exceção `JSONDecodeError`.

Alterado na versão 3.6: *s* agora pode ser um do tipo `bytes` ou `bytearray`. A entrada deve estar codificado como UTF-8, UTF-16 ou UTF-32.

20.2.2 Codificadores e decodificadores

`class` `json.`**`JSONDecoder`**(*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

Decodificador JSON simples

Executa as seguintes traduções na decodificação por padrão:

JSON	Python
object (objeto)	dict
Array	list
string	str
number (int)	int
número (real)	float
true	True
false	False
null	None

Ele também entende `NaN`, `Infinity` e `-Infinity` como seus valores `float` correspondentes, que estão fora da especificação JSON.

object_hook, se especificado, será chamada com o resultado de cada objeto JSON decodificado e seu valor de retorno será usado no lugar do `dict` dado. Isso pode ser usado para fornecer desserializações personalizadas (por exemplo, para oferecer suporte a sugestão para classes JSON-RPC)

object_pairs_hook, se especificado, será chamado com o resultado de cada objeto JSON decodificado com uma lista ordenada de pares. O valor de retorno de *object_pairs_hook* será usado no lugar do `dict`. Este recurso pode ser usado para implementar decodificadores personalizados. Se *object_hook* também for definido, o *object_pairs_hook* terá prioridade.

Alterado na versão 3.1: Adicionado suporte para *object_pairs_hook*.

parse_float, se especificado, será chamada com a string de cada ponto flutuante JSON para ser decodificado. Por padrão, é equivalente a `float(num_str)`. Pode ser usado para qualquer outro tipo de dado ou conversor para ponto flutuante JSON (ex. *decimal.Decimal*).

parse_int, se especificado, será chamada com a string de cada inteiro JSON para ser decodificado. Por padrão, é equivalente a `int(num_str)`. Pode ser usado para qualquer outro tipo de dado ou conversor para inteiro JSON (ex. *float*).

parse_constant, se especificado, será chamada para cada um das seguintes strings: `'-Infinity'`, `'Infinity'`, `'NaN'`. Isso pode ser usado para levantar uma exceção se forem encontrados números JSON inválidos.

Se *strict* for falso (`True` é o padrão), os caracteres de controle serão permitidos dentro das strings. Os caracteres de controle neste contexto são aqueles com códigos de caracteres no intervalo 0-31, incluindo `'\t'` (tab), `'\n'`, `'\r'` and `'\0'`.

Se os dados a serem desserializados não forem um documento JSON válido, será levantada uma exceção *JSONDecodeError*.

Alterado na versão 3.6: Todos os parâmetros agora são *somente-nomeado*.

decode(*s*)
Retorna a representação Python de *s* (uma instância *str* contendo um documento JSON).

JSONDecodeError será levantada se o documento JSON fornecido não for válido.

raw_decode(*s*)
Decodifica um documento JSON a partir de *s* (uma *str* iniciando com um documento JSON) e retornando uma tupla com 2 elementos, a representação Python e o índice em *s* onde o documento finaliza.

Isso pode ser usado para decodificar um documento JSON a partir de uma string que possa ter dados extras ao final.

class json.**JSONEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)
Codificador JSON extensível para estruturas de dados Python.

Por padrão, possui suporte para os seguintes objetos e tipos:

Python	JSON
dict	object (objeto)
list, tuple	Array
str	string
int, float, int- & float-derived Enums	número
True	true
False	false
None	null

Alterado na versão 3.4: Adicionou suporte para classes Enum derivadas de int e float.

Para estender isso para reconhecer outros objetos, crie uma subclasse e implemente o método *default()* com outro método que retorne um objeto serializável para `o` se possível, caso contrário deveria chamar a implementação da superclasse (para levantar *TypeError*).

Se *skipkeys* é falso (o padrão), então será levantada uma *TypeError* ao tentar codificar as chaves que não são *str*, *int*, *float* ou `None`. Se *skipkeys* é verdadeiro, esse itens são simplesmente pulados.

If *ensure_ascii* is true (the default), the output is guaranteed to have all incoming non-ASCII characters escaped. If *ensure_ascii* is false, these characters will be output as-is.

Se *check_circular* é verdadeiro (o padrão), então listas, dicionários, e objetos codificados personalizados serão verificados por referências circulares durante a codificação para prevenir uma recursão infinita (que iria causar uma `OverflowError`). Caso contrário, nenhuma verificação será feita.

Se *allow_nan* for verdadeiro (o padrão), então `NaN`, `Ìnfinity`, and `-Infinity` serão codificados como tal. Esse comportamento não é compatível com a especificação do JSON, mas é consistente com a maioria dos codificadores e decodificadores baseados em JavaScript. Caso contrário, será um `ValueError` para codificar tais pontos flutuantes.

Se *sort_keys* for verdadeiro (padrão: `False`), então a saída dos dicionários serão ordenados pela chave; isto é útil para testes de regressão para certificar-se que as serializações de JSON possam ser comparadas com uma base do dia-a-dia.

Se *indent* for um inteiro não-negativo ou uma string, então elementos de um vetor JSON e membros de objetos terão uma saída formatada com este nível de identação. Um n;ivel de identação 0, negativo ou `""` apenas colocará novas linhas. `None` (o padrão) seleciona a presentação mais compacta. Usando um inteiro positivo a identação terá alguns espaços por nível. Se *indent* for uma string (como `"\t"`), essa string será usada para identar cada nível.

Alterado na versão 3.2: Permite strings para *indent*, além de inteiros.

Se especificado, *separators* deve ser uma tupla `(item_separator, key_separator)`. O padrão é `(', ', ': ')` se *indent* for `None` e `(',', ': ')` caso contrário. Para pegar representação JSON mais compacta, você deve especificar `(',', ':')` para eliminar espaços em branco.

Alterado na versão 3.4: Usa `(',', ': ')` como padrão se *indent* não for `None`.

Se especificado, *default* deve ser uma função para ser chamada para objetos que não podem ser serializados de outra forma. Deve retornar uma versão codificada JSON do objeto ou lançar uma exceção `TypeError`. Se não for especificada, `TypeError` é levantada.

Alterado na versão 3.6: Todos os parâmetros agora são *somente-nomeado*.

default(*o*)
Implemente este método em uma subclasse que retorna um objeto serializável para *o*, ou chame a implementação base (para levantar uma `TypeError`).

Por exemplo, para suporte a iteradores arbitrários, você poderia implementar default dessa forma:

```
def default(self, o):
   try:
       iterable = iter(o)
   except TypeError:
       pass
   else:
       return list(iterable)
   # Let the base class default method raise the TypeError
   return json.JSONEncoder.default(self, o)
```

encode(*o*)
Retorna uma string representando um JSON a partir da estrutura de dados Python, *o*. Por exemplo:

```
>>> json.JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*)
Codifica o objeto dado, *o*, e produz cada representação em string assim que disponível. Por exemplo:

```
for chunk in json.JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

20.2.3 Exceções

exception json.**JSONDecodeError**(*msg*, *doc*, *pos*)
Subclasse de *ValueError* com os seguintes atributos adicionais:

msg
A mensagem de erro não formatada.

doc
O documento JSON sendo analisado.

pos
O índice inicial de *doc* em que a análise falhou.

lineno
A linha correspondente a *pos*.

colno
A coluna correspondente a *pos*.

Novo na versão 3.5.

20.2.4 Conformidade e interoperabilidade padrões

O formato JSON é especificado por **RFC 7159** e por ECMA-404. Esta seção detalha o nível de conformidade deste módulo com a RFC. Para simplificar, as subclasses *JSONEncoder* e *JSONDecoder*, e outros parâmetros além daqueles explicitamente mencionados, não são considerados.

Este módulo não está em conformidade com a RFC de forma estrita, implementando algumas extensões que são JavaScript válidas, mas não JSON válido. Em particular:

- Os valores de números infinitos e NaN são aceitos e produzidos;
- Nomes repetidos em um objeto são aceitos e apenas o valor do último par nome-valor é usado.

Uma vez que a RFC permite que os analisadores compatíveis com RFC aceitem textos de entrada que não sejam compatíveis com RFC, o desserializador deste módulo é tecnicamente compatível com RFC nas configurações padrões.

Codificações de caracteres

A RFC requer que JSON seja representado usando UTF-8, UTF-16 ou UTF-32, com UTF-8 sendo o padrão recomendado para interoperabilidade máxima.

Conforme permitido, embora não exigido, pela RFC, o serializador deste módulo define *ensure_ascii=True* por padrão, escapando a saída para que as strings resultantes contenham apenas caracteres ASCII.

Além do parâmetro *ensure_ascii*, este módulo é definido estritamente em termos de conversão entre objetos Python e *strings Unicode* e, portanto, não aborda diretamente o problema de codificação de caracteres.

A RFC proíbe adicionar uma marca de ordem de byte (do inglês *byte order mark* - BOM) ao início de um texto JSON, e o serializador deste módulo não adiciona um BOM à sua saída. A RFC permite, mas não exige, que os desserializadores JSON ignorem um BOM inicial em sua entrada. O desserializador deste módulo levanta uma *ValueError* quando um BOM inicial está presente.

A RFC não proíbe explicitamente as strings JSON que contêm sequências de bytes que não correspondem a caracteres Unicode válidos (por exemplo, substitutos UTF-16 não emparelhados), mas observa que podem causar problemas de interoperabilidade. Por padrão, este módulo aceita e produz (quando presente no original *str*) pontos de código para tais sequências.

Valores numéricos infinitos e NaN

A RFC não permite a representação de valores infinitos ou numéricos NaN. Apesar disso, por padrão, este módulo aceita e produz `Infinity`, `-Infinity` e `NaN` como se fossem valores literais de número JSON válidos:

```
>>> # Neither of these calls raises an exception, but the results are not valid JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

No serializador, o parâmetro *allow_nan* pode ser usado para alterar esse comportamento. No desserializador, o parâmetro *parse_constant* pode ser usado para alterar esse comportamento.

Nomes repetidos dentro de um objeto

A RFC especifica que os nomes em um objeto JSON devem ser exclusivos, mas não determina como os nomes repetidos em objetos JSON devem ser tratados. Por padrão, este módulo não levanta uma exceção; em vez disso, ele ignora tudo, exceto o último par nome-valor para um determinado nome:

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
```

O parâmetro *object_pairs_hook* pode ser usado para alterar este comportamento.

Valores não object e não array de nível superior

A versão antiga de JSON especificada pela obsoleta **RFC 4627** exige que o valor de nível superior do texto JSON deve ser do tipo object ou array (Python *dict* ou *list*), e não poderia ser dos tipos null, boolean, number, ou string. **RFC 7159** removeu essa restrição, e esse módulo não tem nenhuma implementação que faça essa restrição, seja em seus serializadores, sejam nos desserializadores.

Independentemente, para máxima interoperabilidade, você poderia querer aderir voluntariamente a restrição.

Limitações de implementação

Algumas implementações de desserializadores JSON podem definir limites em:

- o tamanho de textos JSON aceitos
- o máximo de níveis de aninhamentos de objetos e vetores JSON
- o intervalo e a precisão de números JSON
- o conteúdo e o tamanho máximo de strings JSON

Esse módulo não impõe nenhum limite além daqueles já colocados pelas estruturas de dados Python ou pelo interpretador Python em si.

Quando serializando para JSON, tenha cuidado com qualquer limitação nas aplicações que irão consumir seu JSON. Em particular, é comum para números JSON serem desserializados com a precisão dupla definida em IEEE 754, portanto

sujeito aos intervalos de representação e limitações de precisão. Isso é especialmente relevante quando serializando valores Python *int* de magnitude extremamente grande, ou quando serializando instâncias de tipos números "exóticos" como *decimal.Decimal*.

20.2.5 Interface de Linha de Comando

Código-fonte: Lib/json/tool.py

O módulo *json.tool* fornece uma interface de linha de comando simples para validação e embelezamento de saída para objetos JSON.

Se os argumentos opcionais `infile` e `outfile` não forem especificados, *sys.stdin* e *sys.stdout* serão usados respectivamente:

```
$ echo '{"json": "obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

Alterado na versão 3.5: A saída agora está na mesma ordem da entrada. Use a opção *--sort-keys* para ordenar a saída de dicionários alfabeticamente pela chave.

Opções da linha de comando

infile
O arquivo JSON para ser validado ou saída embelezada:

```
$ python -m json.tool mp_films.json
[
    {
        "title": "And Now for Something Completely Different",
        "year": 1971
    },
    {
        "title": "Monty Python and the Holy Grail",
        "year": 1975
    }
]
```

Se *infile* não é especificado, lê de *sys.stdin*.

outfile
Escreve a saída de *infile* para o *outfile* dado. Caso contrário, escreve em *sys.stdout*.

--sort-keys
Ordena a saída de dicionários alfabeticamente pela chave.

Novo na versão 3.5.

-h, --help
Exibe a mensagem de ajuda.

20.3 mailcap — Mailcap file handling

Código Fonte: Lib/mailcap.py

Mailcap files are used to configure how MIME-aware applications such as mail readers and Web browsers react to files with different MIME types. (The name "mailcap" is derived from the phrase "mail capability".) For example, a mailcap file might contain a line like `video/mpeg; xmpeg %s`. Then, if the user encounters an email message or Web document with the MIME type *video/mpeg*, `%s` will be replaced by a filename (usually one belonging to a temporary file) and the **xmpeg** program can be automatically started to view the file.

The mailcap format is documented in **RFC 1524**, "A User Agent Configuration Mechanism For Multimedia Mail Format Information", but is not an Internet standard. However, mailcap files are supported on most Unix systems.

mailcap.**findmatch**(*caps*, *MIMEtype*, *key='view'*, *filename='/dev/null'*, *plist=[]*)

Return a 2-tuple; the first element is a string containing the command line to be executed (which can be passed to *os.system()*), and the second element is the mailcap entry for a given MIME type. If no matching MIME type can be found, `(None, None)` is returned.

key is the name of the field desired, which represents the type of activity to be performed; the default value is 'view', since in the most common case you simply want to view the body of the MIME-typed data. Other possible values might be 'compose' and 'edit', if you wanted to create a new body of the given MIME type or alter the existing body data. See **RFC 1524** for a complete list of these fields.

filename is the filename to be substituted for `%s` in the command line; the default value is `'/dev/null'` which is almost certainly not what you want, so usually you'll override it by specifying a filename.

plist can be a list containing named parameters; the default value is simply an empty list. Each entry in the list must be a string containing the parameter name, an equals sign (`'='`), and the parameter's value. Mailcap entries can contain named parameters like `%{foo}`, which will be replaced by the value of the parameter named 'foo'. For example, if the command line `showpartial %{id} %{number} %{total}` was in a mailcap file, and *plist* was set to `['id=1', 'number=2', 'total=3']`, the resulting command line would be `'showpartial 1 2 3'`.

In a mailcap file, the "test" field can optionally be specified to test some external condition (such as the machine architecture, or the window system in use) to determine whether or not the mailcap line applies. *findmatch()* will automatically check such conditions and skip the entry if the check fails.

Alterado na versão 3.7.16: To prevent security issues with shell metacharacters (symbols that have special effects in a shell command line), `findmatch` will refuse to inject ASCII characters other than alphanumerics and `@+=:,./-_` into the returned command line.

If a disallowed character appears in *filename*, `findmatch` will always return `(None, None)` as if no entry was found. If such a character appears elsewhere (a value in *plist* or in *MIMEtype*), `findmatch` will ignore all mailcap entries which use that value. A *warning* will be raised in either case.

mailcap.**getcaps**()

Returns a dictionary mapping MIME types to a list of mailcap file entries. This dictionary must be passed to the *findmatch()* function. An entry is stored as a list of dictionaries, but it shouldn't be necessary to know the details of this representation.

The information is derived from all of the mailcap files found on the system. Settings in the user's mailcap file `$HOME/.mailcap` will override settings in the system mailcap files `/etc/mailcap`, `/usr/etc/mailcap`, and `/usr/local/etc/mailcap`.

Um exemplo de uso:

```
>>> import mailcap
>>> d = mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='tmp1223')
('xmpeg tmp1223', {'view': 'xmpeg %s'})
```

20.4 mailbox — Manipulate mailboxes in various formats

Código Fonte: Lib/mailbox.py

This module defines two classes, *Mailbox* and *Message*, for accessing and manipulating on-disk mailboxes and the messages they contain. *Mailbox* offers a dictionary-like mapping from keys to messages. *Message* extends the *email.message* module's *Message* class with format-specific state and behavior. Supported mailbox formats are Maildir, mbox, MH, Babyl, and MMDF.

Ver também:

Module *email* Represent and manipulate messages.

20.4.1 Mailbox objects

class mailbox.**Mailbox**

A mailbox, which may be inspected and modified.

The *Mailbox* class defines an interface and is not intended to be instantiated. Instead, format-specific subclasses should inherit from *Mailbox* and your code should instantiate a particular subclass.

The *Mailbox* interface is dictionary-like, with small keys corresponding to messages. Keys are issued by the *Mailbox* instance with which they will be used and are only meaningful to that *Mailbox* instance. A key continues to identify a message even if the corresponding message is modified, such as by replacing it with another message.

Messages may be added to a *Mailbox* instance using the set-like method *add()* and removed using a `del` statement or the set-like methods *remove()* and *discard()*.

Mailbox interface semantics differ from dictionary semantics in some noteworthy ways. Each time a message is requested, a new representation (typically a *Message* instance) is generated based upon the current state of the mailbox. Similarly, when a message is added to a *Mailbox* instance, the provided message representation's contents are copied. In neither case is a reference to the message representation kept by the *Mailbox* instance.

The default *Mailbox* iterator iterates over message representations, not keys as the default dictionary iterator does. Moreover, modification of a mailbox during iteration is safe and well-defined. Messages added to the mailbox after an iterator is created will not be seen by the iterator. Messages removed from the mailbox before the iterator yields them will be silently skipped, though using a key from an iterator may result in a *KeyError* exception if the corresponding message is subsequently removed.

> **Aviso:** Be very cautious when modifying mailboxes that might be simultaneously changed by some other process. The safest mailbox format to use for such tasks is Maildir; try to avoid using single-file formats such as mbox for concurrent writing. If you're modifying a mailbox, you *must* lock it by calling the *lock()* and *unlock()* methods *before* reading any messages in the file or making any changes by adding or deleting a message. Failing to lock the mailbox runs the risk of losing messages or corrupting the entire mailbox.

Mailbox instances have the following methods:

add(*message*)
Add *message* to the mailbox and return the key that has been assigned to it.

Parameter *message* may be a `Message` instance, an `email.message.Message` instance, a string, a byte string, or a file-like object (which should be open in binary mode). If *message* is an instance of the appropriate format-specific `Message` subclass (e.g., if it's an `mboxMessage` instance and this is an `mbox` instance), its format-specific information is used. Otherwise, reasonable defaults for format-specific information are used.

Alterado na versão 3.2: Support for binary input was added.

remove(*key*)
__delitem__(*key*)
discard(*key*)
Delete the message corresponding to *key* from the mailbox.

If no such message exists, a `KeyError` exception is raised if the method was called as `remove()` or `__delitem__()` but no exception is raised if the method was called as `discard()`. The behavior of `discard()` may be preferred if the underlying mailbox format supports concurrent modification by other processes.

__setitem__(*key*, *message*)
Replace the message corresponding to *key* with *message*. Raise a `KeyError` exception if no message already corresponds to *key*.

As with `add()`, parameter *message* may be a `Message` instance, an `email.message.Message` instance, a string, a byte string, or a file-like object (which should be open in binary mode). If *message* is an instance of the appropriate format-specific `Message` subclass (e.g., if it's an `mboxMessage` instance and this is an `mbox` instance), its format-specific information is used. Otherwise, the format-specific information of the message that currently corresponds to *key* is left unchanged.

iterkeys()
keys()
Return an iterator over all keys if called as `iterkeys()` or return a list of keys if called as `keys()`.

itervalues()
__iter__()
values()
Return an iterator over representations of all messages if called as `itervalues()` or `__iter__()` or return a list of such representations if called as `values()`. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

Nota: The behavior of `__iter__()` is unlike that of dictionaries, which iterate over keys.

---

iteritems()
items()
Return an iterator over (*key*, *message*) pairs, where *key* is a key and *message* is a message representation, if called as `iteritems()` or return a list of such pairs if called as `items()`. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

get(*key*, *default=None*)
__getitem__(*key*)
Return a representation of the message corresponding to *key*. If no such message exists, *default* is returned if the method was called as `get()` and a `KeyError` exception is raised if the method was called as `__getitem__()`. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

get_message(*key*)
Return a representation of the message corresponding to *key* as an instance of the appropriate format-specific `Message` subclass, or raise a `KeyError` exception if no such message exists.

get_bytes(*key*)
Return a byte representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists.

Novo na versão 3.2.

get_string(*key*)
Return a string representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists. The message is processed through `email.message.Message` to convert it to a 7bit clean representation.

get_file(*key*)
Return a file-like representation of the message corresponding to *key*, or raise a `KeyError` exception if no such message exists. The file-like object behaves as if open in binary mode. This file should be closed once it is no longer needed.

Alterado na versão 3.2: The file object really is a binary file; previously it was incorrectly returned in text mode. Also, the file-like object now supports the context management protocol: you can use a `with` statement to automatically close it.

Nota: Unlike other representations of messages, file-like representations are not necessarily independent of the `Mailbox` instance that created them or of the underlying mailbox. More specific documentation is provided by each subclass.

__contains__(*key*)
Return `True` if *key* corresponds to a message, `False` otherwise.

__len__()
Return a count of messages in the mailbox.

clear()
Delete all messages from the mailbox.

pop(*key*, *default=None*)
Return a representation of the message corresponding to *key* and delete the message. If no such message exists, return *default*. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

popitem()
Return an arbitrary (*key*, *message*) pair, where *key* is a key and *message* is a message representation, and delete the corresponding message. If the mailbox is empty, raise a `KeyError` exception. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the `Mailbox` instance was initialized.

update(*arg*)
Parameter *arg* should be a *key*-to-*message* mapping or an iterable of (*key*, *message*) pairs. Updates the mailbox so that, for each given *key* and *message*, the message corresponding to *key* is set to *message* as if by using `__setitem__()`. As with `__setitem__()`, each *key* must already correspond to a message in the mailbox or else a `KeyError` exception will be raised, so in general it is incorrect for *arg* to be a `Mailbox` instance.

Nota: Unlike with dictionaries, keyword arguments are not supported.

---

flush()
Write any pending changes to the filesystem. For some *Mailbox* subclasses, changes are always written immediately and *flush()* does nothing, but you should still make a habit of calling this method.

lock()
Acquire an exclusive advisory lock on the mailbox so that other processes know not to modify it. An *ExternalClashError* is raised if the lock is not available. The particular locking mechanisms used depend upon the mailbox format. You should *always* lock the mailbox before making any modifications to its contents.

unlock()
Release the lock on the mailbox, if any.

close()
Flush the mailbox, unlock it if necessary, and close any open files. For some *Mailbox* subclasses, this method does nothing.

Maildir

class mailbox.**Maildir**(*dirname*, *factory=None*, *create=True*)
A subclass of *Mailbox* for mailboxes in Maildir format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *MaildirMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

If *create* is `True` and the *dirname* path exists, it will be treated as an existing maildir without attempting to verify its directory layout.

It is for historical reasons that *dirname* is named as such rather than *path*.

Maildir is a directory-based mailbox format invented for the qmail mail transfer agent and now widely supported by other programs. Messages in a Maildir mailbox are stored in separate files within a common directory structure. This design allows Maildir mailboxes to be accessed and modified by multiple unrelated programs without data corruption, so file locking is unnecessary.

Maildir mailboxes contain three subdirectories, namely: `tmp`, `new`, and `cur`. Messages are created momentarily in the `tmp` subdirectory and then moved to the `new` subdirectory to finalize delivery. A mail user agent may subsequently move the message to the `cur` subdirectory and store information about the state of the message in a special "info" section appended to its file name.

Folders of the style introduced by the Courier mail transfer agent are also supported. Any subdirectory of the main mailbox is considered a folder if `'.'` is the first character in its name. Folder names are represented by *Maildir* without the leading `'.'`. Each folder is itself a Maildir mailbox but should not contain other folders. Instead, a logical nesting is indicated using `'.'` to delimit levels, e.g., "Archived.2005.07".

Nota: The Maildir specification requires the use of a colon (`':'`) in certain message file names. However, some operating systems do not permit this character in file names, If you wish to use a Maildir-like format on such an operating system, you should specify another character to use instead. The exclamation point (`'!'`) is a popular choice. For example:

```
import mailbox
mailbox.Maildir.colon = '!'
```

The `colon` attribute may also be set on a per-instance basis.

Maildir instances have all of the methods of *Mailbox* in addition to the following:

list_folders()
Return a list of the names of all folders.

get_folder(*folder*)
Return a *Maildir* instance representing the folder whose name is *folder*. A *NoSuchMailboxError* exception is raised if the folder does not exist.

add_folder(*folder*)
Create a folder whose name is *folder* and return a *Maildir* instance representing it.

remove_folder(*folder*)
Delete the folder whose name is *folder*. If the folder contains any messages, a *NotEmptyError* exception will be raised and the folder will not be deleted.

clean()
Delete temporary files from the mailbox that have not been accessed in the last 36 hours. The Maildir specification says that mail-reading programs should do this occasionally.

Some *Mailbox* methods implemented by *Maildir* deserve special remarks:

add(*message*)
__setitem__(*key*, *message*)
update(*arg*)

> **Aviso:** These methods generate unique file names based upon the current process ID. When using multiple threads, undetected name clashes may occur and cause corruption of the mailbox unless threads are coordinated to avoid using these methods to manipulate the same mailbox simultaneously.

flush()
All changes to Maildir mailboxes are immediately applied, so this method does nothing.

lock()
unlock()
Maildir mailboxes do not support (or require) locking, so these methods do nothing.

close()
Maildir instances do not keep any open files and the underlying mailboxes do not support locking, so this method does nothing.

get_file(*key*)
Depending upon the host platform, it may not be possible to modify or remove the underlying message while the returned file remains open.

Ver também:

maildir man page from qmail The original specification of the format.

Using maildir format Notes on Maildir by its inventor. Includes an updated name-creation scheme and details on "info" semantics.

maildir man page from Courier Another specification of the format. Describes a common extension for supporting folders.

mbox

class mailbox.**mbox**(*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in mbox format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *mboxMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

The mbox format is the classic format for storing mail on Unix systems. All messages in an mbox mailbox are stored in a single file with the beginning of each message indicated by a line whose first five characters are "From ".

Several variations of the mbox format exist to address perceived shortcomings in the original. In the interest of compatibility, *mbox* implements the original format, which is sometimes referred to as *mboxo*. This means that the *Content-Length* header, if present, is ignored and that any occurrences of "From " at the beginning of a line in a message body are transformed to ">From " when storing the message, although occurrences of ">From " are not transformed to "From " when reading the message.

Some *Mailbox* methods implemented by *mbox* deserve special remarks:

get_file(*key*)

Using the file after calling `flush()` or `close()` on the *mbox* instance may yield unpredictable results or raise an exception.

lock()
unlock()

Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.

Ver também:

mbox man page from qmail A specification of the format and its variations.

mbox man page from tin Another specification of the format, with details on locking.

Configuring Netscape Mail on Unix: Why The Content-Length Format is Bad An argument for using the original mbox format rather than a variation.

"mbox" is a family of several mutually incompatible mailbox formats A history of mbox variations.

MH

class mailbox.**MH**(*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in MH format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *MHMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

MH is a directory-based mailbox format invented for the MH Message Handling System, a mail user agent. Each message in an MH mailbox resides in its own file. An MH mailbox may contain other MH mailboxes (called *folders*) in addition to messages. Folders may be nested indefinitely. MH mailboxes also support *sequences*, which are named lists used to logically group messages without moving them to sub-folders. Sequences are defined in a file called `.mh_sequences` in each folder.

The *MH* class manipulates MH mailboxes, but it does not attempt to emulate all of **mh**'s behaviors. In particular, it does not modify and is not affected by the `context` or `.mh_profile` files that are used by **mh** to store its state and configuration.

MH instances have all of the methods of *Mailbox* in addition to the following:

list_folders()
Return a list of the names of all folders.

get_folder(*folder*)
Return an *MH* instance representing the folder whose name is *folder*. A *NoSuchMailboxError* exception is raised if the folder does not exist.

add_folder(*folder*)
Create a folder whose name is *folder* and return an *MH* instance representing it.

remove_folder(*folder*)
Delete the folder whose name is *folder*. If the folder contains any messages, a *NotEmptyError* exception will be raised and the folder will not be deleted.

get_sequences()
Return a dictionary of sequence names mapped to key lists. If there are no sequences, the empty dictionary is returned.

set_sequences(*sequences*)
Re-define the sequences that exist in the mailbox based upon *sequences*, a dictionary of names mapped to key lists, like returned by *get_sequences()*.

pack()
Rename messages in the mailbox as necessary to eliminate gaps in numbering. Entries in the sequences list are updated correspondingly.

Nota: Already-issued keys are invalidated by this operation and should not be subsequently used.

Some *Mailbox* methods implemented by *MH* deserve special remarks:

remove(*key*)
__delitem__(*key*)
discard(*key*)
These methods immediately delete the message. The MH convention of marking a message for deletion by prepending a comma to its name is not used.

lock()
unlock()
Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls. For MH mailboxes, locking the mailbox means locking the `.mh_sequences` file and, only for the duration of any operations that affect them, locking individual message files.

get_file(*key*)
Depending upon the host platform, it may not be possible to remove the underlying message while the returned file remains open.

flush()
All changes to MH mailboxes are immediately applied, so this method does nothing.

close()
MH instances do not keep any open files, so this method is equivalent to *unlock()*.

Ver também:

nmh - Message Handling System Home page of **nmh**, an updated version of the original **mh**.

MH & nmh: Email for Users & Programmers A GPL-licensed book on **mh** and **nmh**, with some information on the mailbox format.

Babyl

class mailbox.**Babyl**(*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in Babyl format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *BabylMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

Babyl is a single-file mailbox format used by the Rmail mail user agent included with Emacs. The beginning of a message is indicated by a line containing the two characters Control-Underscore (`'\037'`) and Control-L (`'\014'`). The end of a message is indicated by the start of the next message or, in the case of the last message, a line containing a Control-Underscore (`'\037'`) character.

Messages in a Babyl mailbox have two sets of headers, original headers and so-called visible headers. Visible headers are typically a subset of the original headers that have been reformatted or abridged to be more attractive. Each message in a Babyl mailbox also has an accompanying list of *labels*, or short strings that record extra information about the message, and a list of all user-defined labels found in the mailbox is kept in the Babyl options section.

Babyl instances have all of the methods of *Mailbox* in addition to the following:

get_labels()

Return a list of the names of all user-defined labels used in the mailbox.

Nota: The actual messages are inspected to determine which labels exist in the mailbox rather than consulting the list of labels in the Babyl options section, but the Babyl section is updated whenever the mailbox is modified.

Some *Mailbox* methods implemented by *Babyl* deserve special remarks:

get_file(*key*)

In Babyl mailboxes, the headers of a message are not stored contiguously with the body of the message. To generate a file-like representation, the headers and body are copied together into an *io.BytesIO* instance, which has an API identical to that of a file. As a result, the file-like object is truly independent of the underlying mailbox but does not save memory compared to a string representation.

lock()
unlock()

Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.

Ver também:

Format of Version 5 Babyl Files A specification of the Babyl format.

Reading Mail with Rmail The Rmail manual, with some information on Babyl semantics.

MMDF

class mailbox.**MMDF**(*path*, *factory=None*, *create=True*)

A subclass of *Mailbox* for mailboxes in MMDF format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, *MMDFMessage* is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

MMDF is a single-file mailbox format invented for the Multichannel Memorandum Distribution Facility, a mail transfer agent. Each message is in the same form as an mbox message but is bracketed before and after by lines containing four Control-A (`'\001'`) characters. As with the mbox format, the beginning of each message is

indicated by a line whose first five characters are "From ", but additional occurrences of "From " are not transformed to ">From " when storing messages because the extra message separator lines prevent mistaking such occurrences for the starts of subsequent messages.

Some *Mailbox* methods implemented by *MMDF* deserve special remarks:

get_file(*key*)
Using the file after calling `flush()` or `close()` on the *MMDF* instance may yield unpredictable results or raise an exception.

lock()
unlock()
Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.

Ver também:

mmdf man page from tin A specification of MMDF format from the documentation of tin, a newsreader.

MMDF A Wikipedia article describing the Multichannel Memorandum Distribution Facility.

20.4.2 `Message` objects

class mailbox.**Message**(*message=None*)
A subclass of the *email.message* module's *Message*. Subclasses of *mailbox.Message* add mailbox-format-specific state and behavior.

If *message* is omitted, the new instance is created in a default, empty state. If *message* is an *email.message.Message* instance, its contents are copied; furthermore, any format-specific information is converted insofar as possible if *message* is a *Message* instance. If *message* is a string, a byte string, or a file, it should contain an **RFC 2822**-compliant message, which is read and parsed. Files should be open in binary mode, but text mode files are accepted for backward compatibility.

The format-specific state and behaviors offered by subclasses vary, but in general it is only the properties that are not specific to a particular mailbox that are supported (although presumably the properties are specific to a particular mailbox format). For example, file offsets for single-file mailbox formats and file names for directory-based mailbox formats are not retained, because they are only applicable to the original mailbox. But state such as whether a message has been read by the user or marked as important is retained, because it applies to the message itself.

There is no requirement that *Message* instances be used to represent messages retrieved using *Mailbox* instances. In some situations, the time and memory required to generate *Message* representations might not be acceptable. For such situations, *Mailbox* instances also offer string and file-like representations, and a custom message factory may be specified when a *Mailbox* instance is initialized.

MaildirMessage

class mailbox.**MaildirMessage**(*message=None*)
A message with Maildir-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

Typically, a mail user agent application moves all of the messages in the `new` subdirectory to the `cur` subdirectory after the first time the user opens and closes the mailbox, recording that the messages are old whether or not they've actually been read. Each message in `cur` has an "info" section added to its file name to store information about its state. (Some mail readers may also add an "info" section to messages in `new`.) The "info" section may take one of two forms: it may contain "2," followed by a list of standardized flags (e.g., "2,FR") or it may contain "1," followed by so-called experimental information. Standard flags for Maildir messages are as follows:

Flag	Significado	Explanação
D	Draft	Under composition
F	Flagged	Marked as important
P	Passed	Forwarded, resent, or bounced
R	Replied	Replied to
S	Seen	Read
T	Trashed	Marked for subsequent deletion

MaildirMessage instances offer the following methods:

get_subdir()
Return either "new" (if the message should be stored in the `new` subdirectory) or "cur" (if the message should be stored in the `cur` subdirectory).

Nota: A message is typically moved from `new` to `cur` after its mailbox has been accessed, whether or not the message is has been read. A message `msg` has been read if `"S" in msg.get_flags()` is `True`.

---

set_subdir(*subdir*)
Set the subdirectory the message should be stored in. Parameter *subdir* must be either "new" or "cur".

get_flags()
Return a string specifying the flags that are currently set. If the message complies with the standard Maildir format, the result is the concatenation in alphabetical order of zero or one occurrence of each of `'D'`, `'F'`, `'P'`, `'R'`, `'S'`, and `'T'`. The empty string is returned if no flags are set or if "info" contains experimental semantics.

set_flags(*flags*)
Set the flags specified by *flags* and unset all others.

add_flag(*flag*)
Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character. The current "info" is overwritten whether or not it contains experimental information rather than flags.

remove_flag(*flag*)
Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character. If "info" contains experimental information rather than flags, the current "info" is not modified.

get_date()
Return the delivery date of the message as a floating-point number representing seconds since the epoch.

set_date(*date*)
Set the delivery date of the message to *date*, a floating-point number representing seconds since the epoch.

get_info()
Return a string containing the "info" for a message. This is useful for accessing and modifying "info" that is experimental (i.e., not a list of flags).

set_info(*info*)
Set "info" to *info*, which should be a string.

When a *MaildirMessage* instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	*mboxMessage* or *MMDFMessage* state
“cur” subdirectory	O flag
F flag	F flag
R flag	A flag
S flag	R flag
T flag	D flag

When a *MaildirMessage* instance is created based upon an *MHMessage* instance, the following conversions take place:

Resulting state	*MHMessage* state
“cur” subdirectory	“unseen” sequence
“cur” subdirectory and S flag	no “unseen” sequence
F flag	“flagged” sequence
R flag	“replied” sequence

When a *MaildirMessage* instance is created based upon a *BabylMessage* instance, the following conversions take place:

Resulting state	*BabylMessage* state
“cur” subdirectory	“unseen” label
“cur” subdirectory and S flag	no “unseen” label
P flag	“forwarded” or “resent” label
R flag	“answered” label
T flag	“deleted” label

mboxMessage

class mailbox.**mboxMessage**(*message=None*)

A message with mbox-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

Messages in an mbox mailbox are stored together in a single file. The sender’s envelope address and the time of delivery are typically stored in a line beginning with “From ” that is used to indicate the start of a message, though there is considerable variation in the exact format of this data among mbox implementations. Flags that indicate the state of the message, such as whether it has been read or marked as important, are typically stored in *Status* and *X-Status* headers.

Conventional flags for mbox messages are as follows:

Flag	Significado	Explanação
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The “R” and “O” flags are stored in the *Status* header, and the “D”, “F”, and “A” flags are stored in the *X-Status* header. The flags and headers typically appear in the order mentioned.

mboxMessage instances offer the following methods:

get_from()
Return a string representing the "From " line that marks the start of the message in an mbox mailbox. The leading "From " and the trailing newline are excluded.

set_from(*from_*, *time_*=*None*)
Set the "From " line to *from_*, which should be specified without a leading "From " or trailing newline. For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a *time.struct_time* instance, a tuple suitable for passing to *time.strftime()*, or `True` (to use *time.gmtime()*).

get_flags()
Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

set_flags(*flags*)
Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

add_flag(*flag*)
Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

remove_flag(*flag*)
Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an *mboxMessage* instance is created based upon a *MaildirMessage* instance, a "From " line is generated based upon the *MaildirMessage* instance's delivery date, and the following conversions take place:

Resulting state	*MaildirMessage* state
R flag	S flag
O flag	"cur" subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

When an *mboxMessage* instance is created based upon an *MHMessage* instance, the following conversions take place:

Resulting state	*MHMessage* state
R flag and O flag	no "unseen" sequence
O flag	"unseen" sequence
F flag	"flagged" sequence
A flag	"replied" sequence

When an *mboxMessage* instance is created based upon a *BabylMessage* instance, the following conversions take place:

Resulting state	*BabylMessage* state
R flag and O flag	no "unseen" label
O flag	"unseen" label
D flag	"deleted" label
A flag	"answered" label

When a *Message* instance is created based upon an *MMDFMessage* instance, the "From " line is copied and all flags directly correspond:

Resulting state	*MMDFMessage* state
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

MHMessage

class mailbox.**MHMessage**(*message=None*)

A message with MH-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

MH messages do not support marks or flags in the traditional sense, but they do support sequences, which are logical groupings of arbitrary messages. Some mail reading programs (although not the standard **mh** and **nmh**) use sequences in much the same way flags are used with other formats, as follows:

Sequence	Explanação
unseen	Not read, but previously detected by MUA
replied	Replied to
flagged	Marked as important

MHMessage instances offer the following methods:

get_sequences()

Return a list of the names of sequences that include this message.

set_sequences(*sequences*)

Set the list of sequences that include this message.

add_sequence(*sequence*)

Add *sequence* to the list of sequences that include this message.

remove_sequence(*sequence*)

Remove *sequence* from the list of sequences that include this message.

When an *MHMessage* instance is created based upon a *MaildirMessage* instance, the following conversions take place:

Resulting state	*MaildirMessage* state
"unseen" sequence	no S flag
"replied" sequence	R flag
"flagged" sequence	F flag

When an *MHMessage* instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	*mboxMessage* or *MMDFMessage* state
"unseen" sequence	no R flag
"replied" sequence	A flag
"flagged" sequence	F flag

When an *MHMessage* instance is created based upon a *BabylMessage* instance, the following conversions take place:

Resulting state	*BabylMessage* state
“unseen” sequence	“unseen” label
“replied” sequence	“answered” label

BabylMessage

class mailbox.**BabylMessage**(*message=None*)

A message with Babyl-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

Certain message labels, called *attributes*, are defined by convention to have special meanings. The attributes are as follows:

Label	Explanação
unseen	Not read, but previously detected by MUA
deleted	Marked for subsequent deletion
filed	Copied to another file or mailbox
answered	Replied to
forwarded	Forwarded
edited	Modified by the user
resent	Resent

By default, Rmail displays only visible headers. The *BabylMessage* class, though, uses the original headers because they are more complete. Visible headers may be accessed explicitly if desired.

BabylMessage instances offer the following methods:

get_labels()

Return a list of labels on the message.

set_labels(*labels*)

Set the list of labels on the message to *labels*.

add_label(*label*)

Add *label* to the list of labels on the message.

remove_label(*label*)

Remove *label* from the list of labels on the message.

get_visible()

Return an *Message* instance whose headers are the message’s visible headers and whose body is empty.

set_visible(*visible*)

Set the message’s visible headers to be the same as the headers in *message*. Parameter *visible* should be a *Message* instance, an *email.message.Message* instance, a string, or a file-like object (which should be open in text mode).

update_visible()

When a *BabylMessage* instance’s original headers are modified, the visible headers are not automatically modified to correspond. This method updates the visible headers as follows: each visible header with a corresponding original header is set to the value of the original header, each visible header without a corresponding original header is removed, and any of *Date*, *From*, *Reply-To*, *To*, *CC*, and *Subject* that are present in the original headers but not the visible headers are added to the visible headers.

When a *BabylMessage* instance is created based upon a *MaildirMessage* instance, the following conversions take place:

Resulting state	*MaildirMessage* state
“unseen” label	no S flag
“deleted” label	T flag
“answered” label	R flag
“forwarded” label	P flag

When a *BabylMessage* instance is created based upon an *mboxMessage* or *MMDFMessage* instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	*mboxMessage* or *MMDFMessage* state
“unseen” label	no R flag
“deleted” label	D flag
“answered” label	A flag

When a *BabylMessage* instance is created based upon an *MHMessage* instance, the following conversions take place:

Resulting state	*MHMessage* state
“unseen” label	“unseen” sequence
“answered” label	“replied” sequence

MMDFMessage

class mailbox.**MMDFMessage**(*message=None*)

A message with MMDF-specific behaviors. Parameter *message* has the same meaning as with the *Message* constructor.

As with message in an mbox mailbox, MMDF messages are stored with the sender’s address and the delivery date in an initial line beginning with “From “. Likewise, flags that indicate the state of the message are typically stored in *Status* and *X-Status* headers.

Conventional flags for MMDF messages are identical to those of mbox message and are as follows:

Flag	Significado	Explanação
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The “R” and “O” flags are stored in the *Status* header, and the “D”, “F”, and “A” flags are stored in the *X-Status* header. The flags and headers typically appear in the order mentioned.

MMDFMessage instances offer the following methods, which are identical to those offered by *mboxMessage*:

get_from()

Return a string representing the “From ” line that marks the start of the message in an mbox mailbox. The leading “From ” and the trailing newline are excluded.

set_from(*from_*, *time_=None*)

Set the “From ” line to *from_*, which should be specified without a leading “From ” or trailing newline.

For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a `time.struct_time` instance, a tuple suitable for passing to `time.strftime()`, or `True` (to use `time.gmtime()`).

get_flags()
Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

set_flags(*flags*)
Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

add_flag(*flag*)
Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

remove_flag(*flag*)
Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an `MMDFMessage` instance is created based upon a `MaildirMessage` instance, a "From " line is generated based upon the `MaildirMessage` instance's delivery date, and the following conversions take place:

Resulting state	`MaildirMessage` state
R flag	S flag
O flag	"cur" subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

When an `MMDFMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

Resulting state	`MHMessage` state
R flag and O flag	no "unseen" sequence
O flag	"unseen" sequence
F flag	"flagged" sequence
A flag	"replied" sequence

When an `MMDFMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:

Resulting state	`BabylMessage` state
R flag and O flag	no "unseen" label
O flag	"unseen" label
D flag	"deleted" label
A flag	"answered" label

When an `MMDFMessage` instance is created based upon an `mboxMessage` instance, the "From " line is copied and all flags directly correspond:

Resulting state	*mboxMessage* state
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

20.4.3 Exceções

The following exception classes are defined in the *mailbox* module:

exception mailbox.**Error**
The based class for all other module-specific exceptions.

exception mailbox.**NoSuchMailboxError**
Raised when a mailbox is expected but is not found, such as when instantiating a *Mailbox* subclass with a path that does not exist (and with the *create* parameter set to `False`), or when opening a folder that does not exist.

exception mailbox.**NotEmptyError**
Raised when a mailbox is not empty but is expected to be, such as when deleting a folder that contains messages.

exception mailbox.**ExternalClashError**
Raised when some mailbox-related condition beyond the control of the program causes it to be unable to proceed, such as when failing to acquire a lock that another program already holds a lock, or when a uniquely-generated file name already exists.

exception mailbox.**FormatError**
Raised when the data in a file cannot be parsed, such as when an *MH* instance attempts to read a corrupted `.mh_sequences` file.

20.4.4 Exemplos

A simple example of printing the subjects of all messages in a mailbox that seem interesting:

```
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']       # Could possibly be None.
    if subject and 'python' in subject.lower():
        print(subject)
```

To copy all mail from a Babyl mailbox to an MH mailbox, converting all of the format-specific information that can be converted:

```
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

This example sorts mail from several mailing lists into different mailboxes, being careful to avoid mail corruption due to concurrent modification by other programs, mail loss due to interruption of the program, or premature termination due to malformed messages in the mailbox:

```
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = {name: mailbox.mbox('~/email/%s' % name) for name in list_names}
inbox = mailbox.Maildir('~/Maildir', factory=None)

for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.errors.MessageParseError:
        continue                # The message is malformed. Just leave it.

    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # Get mailbox to use
            box = boxes[name]

            # Write copy to disk before removing original.
            # If there's a crash, you might duplicate a message, but
            # that's better than losing a message completely.
            box.lock()
            box.add(message)
            box.flush()
            box.unlock()

            # Remove original message
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
            inbox.unlock()
            break               # Found destination, so stop looking.

for box in boxes.itervalues():
    box.close()
```

20.5 `mimetypes` — Mapeia nomes de arquivos para tipos MIME

Código Fonte: Lib/mimetypes.py

O módulo *mimetypes* converte entre um nome de arquivo ou URL e o tipo MIME associado à extensão do arquivo. As conversões são fornecidas do nome do arquivo para o tipo MIME e da extensão do tipo MIME para o nome do arquivo; codificações não são suportadas para a última conversão.

O módulo fornece uma classe e várias funções convenientes. As funções são a interface normal para este módulo, mas algumas aplicações também podem estar interessadas na classe.

As funções descritas abaixo fornecem a interface principal para este módulo. Se o módulo não foi inicializado, eles chamarão *init()* se confiarem nas informações *init()* configuradas.

mimetypes.**guess_type**(*url*, *strict=True*)

Guess the type of a file based on its filename or URL, given by *url*. The return value is a tuple `(type,`

encoding) where *type* is None if the type can't be guessed (missing or unknown suffix) or a string of the form 'type/subtype', usable for a MIME *content-type* header.

encoding é None para nenhuma codificação ou o nome do programa usado para codificar (por exemplo **compress** ou **gzip**). A codificação é adequada para uso como cabeçalho *Content-Encoding* , **não** como cabeçalho *Content-Transfer-Encoding*. Os mapeamentos são orientados por tabela. Os sufixos de codificação diferenciam maiúsculas de minúsculas; os sufixos de tipo são testados primeiro com maiúsculas e minúsculas e depois sem maiúsculas.

O argumento opcional *strict* é um sinalizados que especifica se alista de tipos MIME conhecidos é limitada apenas aos tipos oficiais registrados na IANA. Quando *strict* é True (o padrão), apenas os tipos IANA são suportados; quando *strict* é False, alguns tipos MIME adicionais não padronizados, mas geralmente usados, também são reconhecidos.

mimetypes.**guess_all_extensions**(*type*, *strict=True*)
Adivinhe as extensões para um arquivo com base em seu tipo MIME, fornecido pelo *tipo*. O valor de retorno é uma lista de cadeias que fornecem todas as extensões possíveis de nome de arquivo, incluindo o ponto ('.'). Não é garantido que as extensões tenham sido associadas a qualquer fluxo de dados específico, mas seriam mapeadas para o tipo MIME *tipo* por *guess_type()*.

O argumento opcional *strict* tem o mesmo significado que com a função *guess_type()*.

mimetypes.**guess_extension**(*type*, *strict=True*)
Guess the extension for a file based on its MIME type, given by *type*. The return value is a string giving a filename extension, including the leading dot ('.'). The extension is not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by *guess_type()*. If no extension can be guessed for *type*, None is returned.

O argumento opcional *strict* tem o mesmo significado que com a função *guess_type()*.

Some additional functions and data items are available for controlling the behavior of the module.

mimetypes.**init**(*files=None*)
Initialize the internal data structures. If given, *files* must be a sequence of file names which should be used to augment the default type map. If omitted, the file names to use are taken from *knownfiles*; on Windows, the current registry settings are loaded. Each file named in *files* or *knownfiles* takes precedence over those named before it. Calling *init()* repeatedly is allowed.

Specifying an empty list for *files* will prevent the system defaults from being applied: only the well-known values will be present from a built-in list.

If *files* is None the internal data structure is completely rebuilt to its initial default value. This is a stable operation and will produce the same results when called multiple times.

Alterado na versão 3.2: Previously, Windows registry settings were ignored.

mimetypes.**read_mime_types**(*filename*)
Load the type map given in the file *filename*, if it exists. The type map is returned as a dictionary mapping filename extensions, including the leading dot ('.'), to strings of the form 'type/subtype'. If the file *filename* does not exist or cannot be read, None is returned.

mimetypes.**add_type**(*type*, *ext*, *strict=True*)
Add a mapping from the MIME type *type* to the extension *ext*. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

When *strict* is True (the default), the mapping will be added to the official MIME types, otherwise to the non-standard ones.

mimetypes.**inited**
Flag indicating whether or not the global data structures have been initialized. This is set to True by *init()*.

mimetypes.**knownfiles**
List of type map file names commonly installed. These files are typically named `mime.types` and are installed in different locations by different packages.

mimetypes.**suffix_map**
Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately.

mimetypes.**encodings_map**
Dictionary mapping filename extensions to encoding types.

mimetypes.**types_map**
Dictionary mapping filename extensions to MIME types.

mimetypes.**common_types**
Dictionary mapping filename extensions to non-standard, but commonly found MIME types.

An example usage of the module:

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

20.5.1 Objetos MimeTypes

The *MimeTypes* class may be useful for applications which may want more than one MIME-type database; it provides an interface similar to the one of the *mimetypes* module.

class mimetypes.**MimeTypes**(*filenames=()*, *strict=True*)
This class represents a MIME-types database. By default, it provides access to the same database as the rest of this module. The initial database is a copy of that provided by the module, and may be extended by loading additional `mime.types`-style files into the database using the *read()* or *readfp()* methods. The mapping dictionaries may also be cleared before loading additional data if the default data is not desired.

The optional *filenames* parameter can be used to cause additional files to be loaded "on top" of the default database.

suffix_map
Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately. This is initially a copy of the global *suffix_map* defined in the module.

encodings_map
Dictionary mapping filename extensions to encoding types. This is initially a copy of the global *encodings_map* defined in the module.

types_map
Tuple containing two dictionaries, mapping filename extensions to MIME types: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by *common_types* and *types_map*.

types_map_inv
Tuple containing two dictionaries, mapping MIME types to a list of filename extensions: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by `common_types` and `types_map`.

guess_extension (*type*, *strict=True*)
Similar to the `guess_extension()` function, using the tables stored as part of the object.

guess_type (*url*, *strict=True*)
Similar to the `guess_type()` function, using the tables stored as part of the object.

guess_all_extensions (*type*, *strict=True*)
Similar to the `guess_all_extensions()` function, using the tables stored as part of the object.

read (*filename*, *strict=True*)
Load MIME information from a file named *filename*. This uses `readfp()` to parse the file.

If *strict* is `True`, information will be added to list of standard types, else to the list of non-standard types.

readfp (*fp*, *strict=True*)
Carrega informações do tipo MIME de um arquivo aberto *fp*. O arquivo precisa estar no formato padrão dos arquivos `mime.types`.

If *strict* is `True`, information will be added to the list of standard types, else to the list of non-standard types.

read_windows_registry (*strict=True*)
Carrega informações do tipo MIME a partir do registro do Windows.

Disponibilidade: Windows.

If *strict* is `True`, information will be added to the list of standard types, else to the list of non-standard types.

Novo na versão 3.2.

20.6 base64 — Codificações de dados em Base16, Base32, Base64, Base85

Código Fonte: Lib/base64.py

This module provides functions for encoding binary data to printable ASCII characters and decoding such encodings back to binary data. It provides encoding and decoding functions for the encodings specified in **RFC 3548**, which defines the Base16, Base32, and Base64 algorithms, and for the de-facto standard Ascii85 and Base85 encodings.

The **RFC 3548** encodings are suitable for encoding binary data so that it can safely sent by email, used as parts of URLs, or included as part of an HTTP POST request. The encoding algorithm is not the same as the **uuencode** program.

There are two interfaces provided by this module. The modern interface supports encoding *bytes-like objects* to ASCII `bytes`, and decoding *bytes-like objects* or strings containing ASCII to `bytes`. Both base-64 alphabets defined in **RFC 3548** (normal, and URL- and filesystem-safe) are supported.

A interface legada não oferece suporte a decodificação de strings, mas fornece funções para codificação e decodificação de e para *objetos arquivo*. Ele oferece suporte a apenas o alfabeto padrão Base64 e adiciona novas linhas a cada 76 caracteres conforme **RFC 2045**. Note que se você estiver procurando por suporte para **RFC 2045** você provavelmente vai querer conferir o pacote `email`.

Alterado na versão 3.3: Strings Unicode exclusivamente ASCII agora são aceitas pelas funções de decodificação da interface moderna.

Alterado na versão 3.4: Quaisquer *objetos bytes ou similares* agora são aceitos por todas as funções de codificação e decodificação neste módulo. Adicionado suporte a ASCII85/Base85.

A interface moderna oferece:

base64.**b64encode**(*s*, *altchars=None*)
Codifica o *objeto bytes ou similar* *s* usando Base64 e retorna o `bytes` codificado.

Os *altchars* opcionais devem ser um *objeto bytes ou similar* de pelo menos comprimento 2 (caracteres adicionais são ignorados) que especifica um alfabeto alternativo para os caracteres + e /. Isso permite que um aplicativo, por exemplo, gerar strings Base64 seguras para URL ou sistema de arquivos. O padrão é `None`, para o qual o alfabeto Base64 padrão é usado.

base64.**b64decode**(*s*, *altchars=None*, *validate=False*)
Decodifica o *objeto bytes ou similar* ou string ASCII *s* codificada em Base64 e retorna o `bytes` decodificado.

Altchars opcionais devem ser um *objeto bytes ou similar* ou string ASCII de pelo menos 2 (caracteres adicionais são ignorados) que especifica o alfabeto alternativo usado ao invés do + e / caracteres.

A `binascii.Error` exception is raised if *s* is incorrectly padded.

If *validate* is `False` (the default), characters that are neither in the normal base-64 alphabet nor the alternative alphabet are discarded prior to the padding check. If *validate* is `True`, these non-alphabet characters in the input result in a `binascii.Error`.

base64.**standard_b64encode**(*s*)
Encode *bytes-like object* *s* using the standard Base64 alphabet and return the encoded `bytes`.

base64.**standard_b64decode**(*s*)
Decode *bytes-like object* or ASCII string *s* using the standard Base64 alphabet and return the decoded `bytes`.

base64.**urlsafe_b64encode**(*s*)
Encode *bytes-like object* *s* using the URL- and filesystem-safe alphabet, which substitutes − instead of + and _ instead of / in the standard Base64 alphabet, and return the encoded `bytes`. The result can still contain =.

base64.**urlsafe_b64decode**(*s*)
Decode *bytes-like object* or ASCII string *s* using the URL- and filesystem-safe alphabet, which substitutes − instead of + and _ instead of / in the standard Base64 alphabet, and return the decoded `bytes`.

base64.**b32encode**(*s*)
Encode the *bytes-like object* *s* using Base32 and return the encoded `bytes`.

base64.**b32decode**(*s*, *casefold=False*, *map01=None*)
Decode the Base32 encoded *bytes-like object* or ASCII string *s* and return the decoded `bytes`.

casefold opcional é uma flag especificando se um alfabeto minúsculo é aceitável como entrada. Por razões de segurança, o padrão é `False`.

RFC 3548 allows for optional mapping of the digit 0 (zero) to the letter O (oh), and for optional mapping of the digit 1 (one) to either the letter I (eye) or letter L (el). The optional argument *map01* when not `None`, specifies which letter the digit 1 should be mapped to (when *map01* is not `None`, the digit 0 is always mapped to the letter O). For security purposes the default is `None`, so that 0 and 1 are not allowed in the input.

A `binascii.Error` is raised if *s* is incorrectly padded or if there are non-alphabet characters present in the input.

base64.**b16encode**(*s*)
Encode the *bytes-like object* *s* using Base16 and return the encoded `bytes`.

base64.**b16decode**(*s*, *casefold=False*)
Decode the Base16 encoded *bytes-like object* or ASCII string *s* and return the decoded `bytes`.

casefold opcional é uma flag especificando se um alfabeto minúsculo é aceitável como entrada. Por razões de segurança, o padrão é `False`.

A *binascii.Error* is raised if *s* is incorrectly padded or if there are non-alphabet characters present in the input.

base64.**a85encode**(*b*, ***, *foldspaces=False*, *wrapcol=0*, *pad=False*, *adobe=False*)
Encode the *bytes-like object* *b* using Ascii85 and return the encoded *bytes*.

foldspaces is an optional flag that uses the special short sequence 'y' instead of 4 consecutive spaces (ASCII 0x20) as supported by 'btoa'. This feature is not supported by the "standard" Ascii85 encoding.

wrapcol controls whether the output should have newline (`b'\n'`) characters added to it. If this is non-zero, each output line will be at most this many characters long.

pad controls whether the input is padded to a multiple of 4 before encoding. Note that the `btoa` implementation always pads.

adobe controls whether the encoded byte sequence is framed with <~ and ~>, which is used by the Adobe implementation.

Novo na versão 3.4.

base64.**a85decode**(*b*, ***, *foldspaces=False*, *adobe=False*, *ignorechars=b' \t\n\r\v'*)
Decode the Ascii85 encoded *bytes-like object* or ASCII string *b* and return the decoded *bytes*.

foldspaces is a flag that specifies whether the 'y' short sequence should be accepted as shorthand for 4 consecutive spaces (ASCII 0x20). This feature is not supported by the "standard" Ascii85 encoding.

adobe controla se a entrada está no formato Adobe Ascii85 (ou seja, cercada por <~ e ~>).

ignorechars should be a *bytes-like object* or ASCII string containing characters to ignore from the input. This should only contain whitespace characters, and by default contains all whitespace characters in ASCII.

Novo na versão 3.4.

base64.**b85encode**(*b*, *pad=False*)
Encode the *bytes-like object* *b* using base85 (as used in e.g. git-style binary diffs) and return the encoded *bytes*.

If *pad* is true, the input is padded with `b'\0'` so its length is a multiple of 4 bytes before encoding.

Novo na versão 3.4.

base64.**b85decode**(*b*)
Decode the base85-encoded *bytes-like object* or ASCII string *b* and return the decoded *bytes*. Padding is implicitly removed, if necessary.

Novo na versão 3.4.

A interface legada:

base64.**decode**(*input*, *output*)
Decode the contents of the binary *input* file and write the resulting binary data to the *output* file. *input* and *output* must be *file objects*. *input* will be read until `input.readline()` returns an empty bytes object.

base64.**decodebytes**(*s*)
Decode the *bytes-like object* *s*, which must contain one or more lines of base64 encoded data, and return the decoded *bytes*.

Novo na versão 3.1.

base64.**decodestring**(*s*)
Deprecated alias of *decodebytes()*.

Obsoleto desde a versão 3.1.

base64.**encode**(*input*, *output*)
Encode the contents of the binary *input* file and write the resulting base64 encoded data to the *output* file. *input* and *output* must be *file objects*. *input* will be read until `input.read()` returns an empty bytes object. *encode()* inserts a newline character (`b'\n'`) after every 76 bytes of the output, as well as ensuring that the output always ends with a newline, as per **RFC 2045** (MIME).

base64.**encodebytes**(*s*)
Encode the *bytes-like object* *s*, which can contain arbitrary binary data, and return *bytes* containing the base64-encoded data, with newlines (`b'\n'`) inserted after every 76 bytes of output, and ensuring that there is a trailing newline, as per **RFC 2045** (MIME).

Novo na versão 3.1.

base64.**encodestring**(*s*)
Deprecated alias of *encodebytes()*.

Obsoleto desde a versão 3.1.

Um exemplo de uso do módulo:

```
>>> import base64
>>> encoded = base64.b64encode(b'data to be encoded')
>>> encoded
b'ZGF0YSB0byBiZSBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'
```

Ver também:

Módulo *binascii* Módulo de suporte contendo conversões ASCII para binário e binário para ASCII.

RFC 1521 - MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format o
Section 5.2, "Base64 Content-Transfer-Encoding," provides the definition of the base64 encoding.

20.7 binhex — Codifica e decodifica arquivos binhex4

Código-fonte: Lib/binhex.py

Este módulo codifica e decodifica arquivos no formato binhex4, um formato que permite a representação de arquivos Macintosh em ASCII. Apenas a bifurcação de dados é manipulada.

O módulo *binhex* define as seguintes funções:

binhex.**binhex**(*input*, *output*)
Converte um arquivo binário com o nome de arquivo *input* para o arquivo binhex *output*. O parâmetro *output* pode ser um nome de arquivo ou um objeto semelhante a um arquivo (qualquer objeto que suporte um método `write()` e `close()`).

binhex.**hexbin**(*input*, *output*)
Decodifica um arquivo binhex *input*. *input* pode ser um nome de arquivo ou um objeto de arquivo com suporte aos métodos `read()` e `close()`. O arquivo resultante é gravado em um arquivo chamado *output*, a menos que o argumento seja `None`, caso em que o nome do arquivo de saída é lido a partir do arquivo binhex.

A seguinte exceção também está definida:

exception binhex.**Error**
Exceção levantada quando algo não pode ser codificado usando o formato binhex (por exemplo, um nome de arquivo é muito longo para caber no campo de nome de arquivo) ou quando a entrada não consiste em dados binhex corretamente codificados.

Ver também:

Módulo ***binascii*** Módulo de suporte contendo conversões ASCII para binário e binário para ASCII.

20.7.1 Notas

Existe uma interface alternativa, mais poderosa para o codificador e o decodificador, veja a fonte para obter detalhes.

Se você codificar ou decodificar arquivos de texto em plataformas que não sejam Macintosh, elas ainda usarão a antiga convenção de linha do Macintosh (carriage-return como fim de linha).

20.8 binascii — Converte entre binário e ASCII

The *binascii* module contains a number of methods to convert between binary and various ASCII-encoded binary representations. Normally, you will not use these functions directly but use wrapper modules like *uu*, *base64*, or *binhex* instead. The *binascii* module contains low-level functions written in C for greater speed that are used by the higher-level modules.

Nota: a2b_* functions accept Unicode strings containing only ASCII characters. Other functions only accept *bytes-like objects* (such as *bytes*, *bytearray* and other objects that support the buffer protocol).

Alterado na versão 3.3: ASCII-only unicode strings are now accepted by the a2b_* functions.

The *binascii* module defines the following functions:

binascii.**a2b_uu**(*string*)
Convert a single line of uuencoded data back to binary and return the binary data. Lines normally contain 45 (binary) bytes, except for the last line. Line data may be followed by whitespace.

binascii.**b2a_uu**(*data*, *, *backtick=False*)
Convert binary data to a line of ASCII characters, the return value is the converted line, including a newline char. The length of *data* should be at most 45. If *backtick* is true, zeros are represented by '`' instead of spaces.

Alterado na versão 3.7: Adicionado o parâmetro *backtick*.

binascii.**a2b_base64**(*string*)
Convert a block of base64 data back to binary and return the binary data. More than one line may be passed at a time.

binascii.**b2a_base64**(*data*, *, *newline=True*)
Convert binary data to a line of ASCII characters in base64 coding. The return value is the converted line, including a newline char if *newline* is true. The output of this function conforms to **RFC 3548**.

Alterado na versão 3.6: Adicionado o parâmetro *newline*.

binascii.**a2b_qp**(*data*, *header=False*)
Convert a block of quoted-printable data back to binary and return the binary data. More than one line may be passed at a time. If the optional argument *header* is present and true, underscores will be decoded as spaces.

binascii.**b2a_qp**(*data*, *quotetabs=False*, *istext=True*, *header=False*)
Convert binary data to a line(s) of ASCII characters in quoted-printable encoding. The return value is the converted line(s). If the optional argument *quotetabs* is present and true, all tabs and spaces will be encoded. If the optional argument *istext* is present and true, newlines are not encoded but trailing whitespace will be encoded. If the optional argument *header* is present and true, spaces will be encoded as underscores per **RFC 1522**. If the optional argument *header* is present and false, newline characters will be encoded as well; otherwise linefeed conversion might corrupt the binary data stream.

binascii.**a2b_hqx**(*string*)
Convert binhex4 formatted ASCII data to binary, without doing RLE-decompression. The string should contain a complete number of binary bytes, or (in case of the last portion of the binhex4 data) have the remaining bits zero.

binascii.**rledecode_hqx**(*data*)
Perform RLE-decompression on the data, as per the binhex4 standard. The algorithm uses `0x90` after a byte as a repeat indicator, followed by a count. A count of `0` specifies a byte value of `0x90`. The routine returns the decompressed data, unless data input data ends in an orphaned repeat indicator, in which case the *Incomplete* exception is raised.

Alterado na versão 3.2: Accept only bytestring or bytearray objects as input.

binascii.**rlecode_hqx**(*data*)
Perform binhex4 style RLE-compression on *data* and return the result.

binascii.**b2a_hqx**(*data*)
Perform hexbin4 binary-to-ASCII translation and return the resulting string. The argument should already be RLE-coded, and have a length divisible by 3 (except possibly the last fragment).

binascii.**crc_hqx**(*data*, *value*)
Compute a 16-bit CRC value of *data*, starting with *value* as the initial CRC, and return the result. This uses the CRC-CCITT polynomial $x^{16} + x^{12} + x^5 + 1$, often represented as 0x1021. This CRC is used in the binhex4 format.

binascii.**crc32**(*data*[, *value*])
Compute CRC-32, the 32-bit checksum of *data*, starting with an initial CRC of *value*. The default initial CRC is zero. The algorithm is consistent with the ZIP file checksum. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm. Use as follows:

```
print(binascii.crc32(b"hello world"))
# Or, in two pieces:
crc = binascii.crc32(b"hello")
crc = binascii.crc32(b" world", crc)
print('crc32 = {:#010x}'.format(crc))
```

Alterado na versão 3.0: The result is always unsigned. To generate the same numeric value across all Python versions and platforms, use `crc32(data) & 0xffffffff`.

binascii.**b2a_hex**(*data*)
binascii.**hexlify**(*data*)
Return the hexadecimal representation of the binary *data*. Every byte of *data* is converted into the corresponding 2-digit hex representation. The returned bytes object is therefore twice as long as the length of *data*.

Similar functionality (but returning a text string) is also conveniently accessible using the *bytes.hex()* method.

binascii.**a2b_hex**(*hexstr*)
binascii.**unhexlify**(*hexstr*)
Return the binary data represented by the hexadecimal string *hexstr*. This function is the inverse of *b2a_hex()*. *hexstr* must contain an even number of hexadecimal digits (which can be upper or lower case), otherwise an *Error* exception is raised.

Similar functionality (accepting only text string arguments, but more liberal towards whitespace) is also accessible using the *bytes.fromhex()* class method.

exception binascii.**Error**
Exception raised on errors. These are usually programming errors.

exception binascii.**Incomplete**
Exception raised on incomplete data. These are usually not programming errors, but may be handled by reading a little more data and trying again.

Ver também:

Módulo ***base64*** Support for RFC compliant base64-style encoding in base 16, 32, 64, and 85.

Module ***binhex*** Support for the binhex format used on the Macintosh.

Module ***uu*** Support for UU encoding used on Unix.

Módulo ***quopri*** Support for quoted-printable encoding used in MIME email messages.

20.9 quopri — Codifica e decodifica dados MIME imprimidos entre aspas

Código-fonte: Lib/quopri.py

Este módulo realiza codificação e decodificação de transporte imprimida entre aspas, como definido em **RFC 1521**: "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies". A codificação imprimida entre aspas é projetada para dados em que há relativamente poucos caracteres não imprimíveis; o esquema de codificação base64 disponível através do módulo *base64* é mais compacto se existirem muitos desses caracteres, como no envio de um arquivo gráfico.

quopri.**decode**(*input*, *output*, *header=False*)
Decodifica o conteúdo do arquivo *input* e escreve os dados binários decodificados resultantes no arquivo *output*. * input * e *output* devem ser *objetos de arquivos binários*. Se o argumento opcional *header* estiver presente e for true, o sublinhado será decodificado como espaço. Isso é usado para decodificar cabeçalhos codificados em "Q", conforme descrito em **RFC 1522**: "MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text".

quopri.**encode**(*input*, *output*, *quotetabs*, *header=False*)
Codifica o conteúdo do arquivo *input* e grava os dados imprimíveis entre aspas resultantes no arquivo *output*. *input* e *output* devem ser *objetos de arquivos binários*. *quotetabs*, um sinalizador não opcional que controla a codificação de espaços e tabulações incorporados; quando true, codifica esses espaços em branco incorporados e, quando false, os deixa sem codificação. Observe que os espaços e tabulações que aparecem no final das linhas são sempre codificados, conforme **RFC 1521**. *header* é um sinalizador que controla se os espaços são codificados como sublinhados, conforme **RFC 1522**.

quopri.**decodestring**(*s*, *header=False*)
Como *decode()*, exceto pelo fato de aceitar uma fonte *bytes* e retornar o correspondente decodificado *bytes*.

quopri.**encodestring**(*s*, *quotetabs=False*, *header=False*)
Como *encode()*, exceto pelo fato de aceitar uma fonte *bytes* e retornar o *bytes* codificado correspondente. Por padrão, envia um valor `False` para o parâmetro *quotetabs* da função *encode()*.

Ver também:

Módulo ***base64*** Codifica e decodifica dados de base64 MIME

20.10 uu — Codifica e decodifica arquivos uuencode

Código Fonte: Lib/uu.py

Este módulo codifica e decodifica arquivos no formato uuencode, permitindo que dados binários arbitrários sejam transferidos por conexões somente ASCII. Sempre que um argumento de arquivo é esperado, os métodos aceitam um objeto arquivo ou similar. Para compatibilidade com versões anteriores, uma string contendo um nome de caminho também é aceita, e o arquivo correspondente será aberto para leitura e gravação; o nome do caminho `'-'` é entendido como a entrada ou saída padrão. No entanto, essa interface foi descontinuada; é melhor para o chamador abrir o próprio arquivo e ter certeza de que, quando necessário, o modo é `'rb'` ou `'wb'` no Windows.

Este código foi contribuído por Lance Ellinghouse e modificado por Jack Jansen.

O módulo :mod: *uu* define as seguintes funções:

uu.**encode**(*in_file*, *out_file*, *name=None*, *mode=None*, *, *backtick=False*)
: Arquivo uuencode *in_file* no arquivo *out_file*. O arquivo uuencoded terá o cabeçalho especificando *name* e *mode* como os padrões para os resultados da decodificação do arquivo. Os padrões padrão são obtidos de *in_file*, ou `'-'` e `0o666` respectivamente. Se *backtick* for verdadeiro, zeros são representados por `` '`' `` ao invés de espaços.

  Alterado na versão 3.7: Adicionado o parâmetro *backtick*.

uu.**decode**(*in_file*, *out_file=None*, *mode=None*, *quiet=False*)
: Esta chamada decodifica o arquivo uuencoded *in_file* colocando o resultado no arquivo *out_file*. Se *out_file* for um nome de caminho, *mode* será usado para definir os bits de permissão se o arquivo precisar ser criado. Os padrões para *out_file* e *mode* são retirados do cabeçalho uuencode. Porém, se o arquivo especificado no cabeçalho já existir, uma *uu.Error* é levantada.

  decode() pode imprimir um aviso de erro padrão se a entrada foi produzida por um uuencoder incorreto e o Python pôde se recuperar desse erro. Definir *quiet* com um valor verdadeiro silencia este aviso.

exception uu.**Error**
: Subclasse de *Exception*, isso pode ser levantada por *uu.decode()* em várias situações, como descrito acima, mas também incluindo um cabeçalho mal formatado ou arquivo de entrada truncado.

Ver também:

Módulo *binascii* Módulo de suporte contendo conversões ASCII para binário e binário para ASCII.

CAPÍTULO 21

Ferramentas de Processamento de Markup Estruturado

O Python suporta uma variedade de módulos para trabalhar com vários formatos de marcação de dados estruturados. Isso inclui módulos para trabalhar com o Standard Generalized Markup Language (SGML) e o Hypertext Markup Language (HTML) e várias interfaces para trabalhar com o XML (Extensible Markup Language).

21.1 `html` — Suporte HTML(HyperText Markup Language)

Código fonte :fonte:‘Lib/html/__init__.py‘

Este módulo define utilitários para manipular HTML.

`html.`**`escape`**(*s*, *quote=True*)
Converte os caracteres `&`, < e > na string *s* para sequências seguras em HTML. Use se necessitar mostrar texto que possa conter estes caracteres no HTML. Se o flag opcional *quote* é true, os caracteres (`"`) e (`'`) também são convertidos; isso auxilia na inclusão de valores delimitados por aspas num atributo HTML, como em `<a href=". . ">`.

Novo na versão 3.2.

`html.`**`unescape`**(*s*)
Converte todas as referências de caracteres numéricos e nomeados (ex. `>`, `&#62;`, `&#x3e;`) na string *s* para caracteres Unicode correspondentes. Essa função usa as regras definidas pelo padrão HTML 5 para referências de caracteres, sejam válidas ou inválidas, e a *lista de referência de caracteres nomeados do HTML 5*.

Novo na versão 3.4.

Sub módulos no pacote `html` são:

- *`html.parser`* – analisador HTML/XHTML com modo de análise branda
- *`html.entities`* – definições das entidade HTML

21.2 `html.parser` — Simple HTML and XHTML parser

Código Fonte: Lib/html/parser.py

This module defines a class *HTMLParser* which serves as the basis for parsing text files formatted in HTML (HyperText Mark-up Language) and XHTML.

class `html.parser.`**HTMLParser**(*, *convert_charrefs=True*)

Create a parser instance able to parse invalid markup.

If *convert_charrefs* is `True` (the default), all character references (except the ones in `script`/`style` elements) are automatically converted to the corresponding Unicode characters.

An *HTMLParser* instance is fed HTML data and calls handler methods when start tags, end tags, text, comments, and other markup elements are encountered. The user should subclass *HTMLParser* and override its methods to implement the desired behavior.

This parser does not check that end tags match start tags or call the end-tag handler for elements which are closed implicitly by closing an outer element.

Alterado na versão 3.4: *convert_charrefs* keyword argument added.

Alterado na versão 3.5: The default value for argument *convert_charrefs* is now `True`.

21.2.1 Example HTML Parser Application

As a basic example, below is a simple HTML parser that uses the *HTMLParser* class to print out start tags, end tags, and data as they are encountered:

```
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Encountered a start tag:", tag)

    def handle_endtag(self, tag):
        print("Encountered an end tag :", tag)

    def handle_data(self, data):
        print("Encountered some data  :", data)

parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
            '<body><h1>Parse me!</h1></body></html>')
```

The output will then be:

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data  : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data  : Parse me!
```

(continua na próxima página)

(continuação da página anterior)

```
Encountered an end tag : h1
Encountered an end tag : body
Encountered an end tag : html
```

21.2.2 `HTMLParser` Methods

HTMLParser instances have the following methods:

`HTMLParser.`**`feed`**(*data*)
Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is buffered until more data is fed or *close()* is called. *data* must be *str*.

`HTMLParser.`**`close`**()
Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call the *HTMLParser* base class method *close()*.

`HTMLParser.`**`reset`**()
Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.

`HTMLParser.`**`getpos`**()
Return current line number and offset.

`HTMLParser.`**`get_starttag_text`**()
Return the text of the most recently opened start tag. This should not normally be needed for structured processing, but may be useful in dealing with HTML "as deployed" or for re-generating input with minimal changes (whitespace between attributes can be preserved, etc.).

The following methods are called when data or markup elements are encountered and they are meant to be overridden in a subclass. The base class implementations do nothing (except for *handle_startendtag()*):

`HTMLParser.`**`handle_starttag`**(*tag*, *attrs*)
This method is called to handle the start of a tag (e.g. `<div id="main">`).

The *tag* argument is the name of the tag converted to lower case. The *attrs* argument is a list of `(name, value)` pairs containing the attributes found inside the tag's <> brackets. The *name* will be translated to lower case, and quotes in the *value* have been removed, and character and entity references have been replaced.

For instance, for the tag `<A HREF="https://www.cwi.nl/">`, this method would be called as `handle_starttag('a', [('href', 'https://www.cwi.nl/')])`.

All entity references from *html.entities* are replaced in the attribute values.

`HTMLParser.`**`handle_endtag`**(*tag*)
This method is called to handle the end tag of an element (e.g. `</div>`).

The *tag* argument is the name of the tag converted to lower case.

`HTMLParser.`**`handle_startendtag`**(*tag*, *attrs*)
Similar to *handle_starttag()*, but called when the parser encounters an XHTML-style empty tag (`<img ... />`). This method may be overridden by subclasses which require this particular lexical information; the default implementation simply calls *handle_starttag()* and *handle_endtag()*.

`HTMLParser.`**`handle_data`**(*data*)
This method is called to process arbitrary data (e.g. text nodes and the content of `<script>...</script>` and `<style>...</style>`).

HTMLParser.**handle_entityref**(*name*)
This method is called to process a named character reference of the form `&name;` (e.g. `>`), where *name* is a general entity reference (e.g. `'gt'`). This method is never called if *convert_charrefs* is `True`.

HTMLParser.**handle_charref**(*name*)
This method is called to process decimal and hexadecimal numeric character references of the form `&#NNN;` and `&#xNNN;`. For example, the decimal equivalent for `>` is `&#62;`, whereas the hexadecimal is `&#x3E;`; in this case the method will receive `'62'` or `'x3E'`. This method is never called if *convert_charrefs* is `True`.

HTMLParser.**handle_comment**(*data*)
This method is called when a comment is encountered (e.g. `<!--comment-->`).

For example, the comment `<!-- comment -->` will cause this method to be called with the argument `' comment '`.

The content of Internet Explorer conditional comments (condcoms) will also be sent to this method, so, for `<!--[if IE 9]>IE9-specific content<![endif]-->`, this method will receive `'[if IE 9]>IE9-specific content<![endif]'`.

HTMLParser.**handle_decl**(*decl*)
This method is called to handle an HTML doctype declaration (e.g. `<!DOCTYPE html>`).

The *decl* parameter will be the entire contents of the declaration inside the `<!...>` markup (e.g. `'DOCTYPE html'`).

HTMLParser.**handle_pi**(*data*)
Method called when a processing instruction is encountered. The *data* parameter will contain the entire processing instruction. For example, for the processing instruction `<?proc color='red'>`, this method would be called as `handle_pi("proc color='red'")`. It is intended to be overridden by a derived class; the base class implementation does nothing.

Nota: The *HTMLParser* class uses the SGML syntactic rules for processing instructions. An XHTML processing instruction using the trailing `'?'` will cause the `'?'` to be included in *data*.

HTMLParser.**unknown_decl**(*data*)
This method is called when an unrecognized declaration is read by the parser.

The *data* parameter will be the entire contents of the declaration inside the `<![...]>` markup. It is sometimes useful to be overridden by a derived class. The base class implementation does nothing.

21.2.3 Exemplos

The following class implements a parser that will be used to illustrate more examples:

```
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Start tag:", tag)
        for attr in attrs:
            print("     attr:", attr)

    def handle_endtag(self, tag):
        print("End tag  :", tag)
```

(continua na próxima página)

(continuação da página anterior)

```
    def handle_data(self, data):
        print("Data     :", data)

    def handle_comment(self, data):
        print("Comment  :", data)

    def handle_entityref(self, name):
        c = chr(name2codepoint[name])
        print("Named ent:", c)

    def handle_charref(self, name):
        if name.startswith('x'):
            c = chr(int(name[1:], 16))
        else:
            c = chr(int(name))
        print("Num ent  :", c)

    def handle_decl(self, data):
        print("Decl     :", data)

parser = MyHTMLParser()
```

Parsing a doctype:

```
>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" '
...             '"http://www.w3.org/TR/html4/strict.dtd">')
Decl     : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
↪html4/strict.dtd"
```

Parsing an element with a few attributes and a title:

```
>>> parser.feed('<img src="python-logo.png" alt="The Python logo">')
Start tag: img
     attr: ('src', 'python-logo.png')
     attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data     : Python
End tag  : h1
```

The content of `script` and `style` elements is returned as is, without further parsing:

```
>>> parser.feed('<style type="text/css">#python { color: green }</style>')
Start tag: style
     attr: ('type', 'text/css')
Data     : #python { color: green }
End tag  : style

>>> parser.feed('<script type="text/javascript">'
...             'alert("<strong>hello!</strong>");</script>')
Start tag: script
     attr: ('type', 'text/javascript')
Data     : alert("<strong>hello!</strong>");
End tag  : script
```

Parsing comments:

```
>>> parser.feed('<!-- a comment -->'
...             '<!--[if IE 9]>IE-specific content<![endif]-->')
Comment  :  a comment
Comment  : [if IE 9]>IE-specific content<![endif]
```

Parsing named and numeric character references and converting them to the correct char (note: these 3 references are all equivalent to `'>'`):

```
>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent  : >
Num ent  : >
```

Feeding incomplete chunks to *feed()* works, but *handle_data()* might be called more than once (unless *convert_charrefs* is set to `True`):

```
>>> for chunk in ['<sp', 'an>buff', 'ered ', 'text</s', 'pan>']:
...     parser.feed(chunk)
...
Start tag: span
Data     : buff
Data     : ered
Data     : text
End tag  : span
```

Parsing invalid HTML (e.g. unquoted attributes) also works:

```
>>> parser.feed('<p><a class=link href=#main>tag soup</p ></a>')
Start tag: p
Start tag: a
     attr: ('class', 'link')
     attr: ('href', '#main')
Data     : tag soup
End tag  : p
End tag  : a
```

21.3 `html.entities` — Definições de entidades gerais de HTML

Código-fonte: Lib/html/entities.py

Esse módulo define quatro dicionários, *html5*, *name2codepoint*, *codepoint2name* e *entitydefs*.

`html.entities.`**`html5`**
: Um dicionário que mapeia referências de caracteres nomeados em HTML5[1] para os caracteres Unicode equivalentes, por exemplo, `html5['gt;'] == '>'`. Note que o caractere de ponto e vírgula final está incluído no nome (por exemplo, `'gt;'`), entretanto alguns dos nomes são aceitos pelo padrão mesmo sem o ponto e vírgula: neste caso o nome está presente com e sem o `';'`. Veja também *html.unescape()*.

  Novo na versão 3.3.

`html.entities.`**`entitydefs`**
: Um dicionário que mapeia as definições de entidade XHTML 1.0 para seu texto substituto em ISO Latin-1.

[1] Veja https://www.w3.org/TR/html5/syntax.html#named-character-references

`html.entities.`**`name2codepoint`**
Um dicionário que mapeia nomes de entidades HTML para os pontos de código Unicode.

`html.entities.`**`codepoint2name`**
Um dicionário que mapeia pontos de código Unicode para nomes de entidades HTML.

21.4 XML Processing Modules

Source code: Lib/xml/

Python's interfaces for processing XML are grouped in the `xml` package.

Aviso: The XML modules are not secure against erroneous or maliciously constructed data. If you need to parse untrusted or unauthenticated data see the *XML vulnerabilities* and *The defusedxml and defusedexpat Packages* sections.

It is important to note that modules in the `xml` package require that there be at least one SAX-compliant XML parser available. The Expat parser is included with Python, so the `xml.parsers.expat` module will always be available.

The documentation for the `xml.dom` and `xml.sax` packages are the definition of the Python bindings for the DOM and SAX interfaces.

The XML handling submodules are:

- `xml.etree.ElementTree`: the ElementTree API, a simple and lightweight XML processor
- `xml.dom`: the DOM API definition
- `xml.dom.minidom`: a minimal DOM implementation
- `xml.dom.pulldom`: support for building partial DOM trees
- `xml.sax`: SAX2 base classes and convenience functions
- `xml.parsers.expat`: the Expat parser binding

21.4.1 XML vulnerabilities

The XML processing modules are not secure against maliciously constructed data. An attacker can abuse XML features to carry out denial of service attacks, access local files, generate network connections to other machines, or circumvent firewalls.

The following table gives an overview of the known attacks and whether the various modules are vulnerable to them.

tipo	sax	etree	minidom	pulldom	xmlrpc
billion laughs	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)
quadratic blowup	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)
external entity expansion	Safe (5)	Safe (2)	Safe (3)	Safe (5)	Safe (4)
DTD retrieval	Safe (5)	Safe	Safe	Safe (5)	Safe
decompression bomb	Safe	Safe	Safe	Safe	**Vulnerable**

1. Expat 2.4.1 and newer is not vulnerable to the "billion laughs" and "quadratic blowup" vulnerabilities. Items still listed as vulnerable due to potential reliance on system-provided libraries. Check `pyexpat.EXPAT_VERSION`.

2. *xml.etree.ElementTree* doesn't expand external entities and raises a `ParserError` when an entity occurs.
3. *xml.dom.minidom* doesn't expand external entities and simply returns the unexpanded entity verbatim.
4. `xmlrpclib` doesn't expand external entities and omits them.
5. Since Python 3.7.1, external general entities are no longer processed by default.

billion laughs / exponential entity expansion The Billion Laughs attack – also known as exponential entity expansion – uses multiple levels of nested entities. Each entity refers to another entity several times, and the final entity definition contains a small string. The exponential expansion results in several gigabytes of text and consumes lots of memory and CPU time.

quadratic blowup entity expansion A quadratic blowup attack is similar to a Billion Laughs attack; it abuses entity expansion, too. Instead of nested entities it repeats one large entity with a couple of thousand chars over and over again. The attack isn't as efficient as the exponential case but it avoids triggering parser countermeasures that forbid deeply-nested entities.

external entity expansion Entity declarations can contain more than just text for replacement. They can also point to external resources or local files. The XML parser accesses the resource and embeds the content into the XML document.

DTD retrieval Some XML libraries like Python's *xml.dom.pulldom* retrieve document type definitions from remote or local locations. The feature has similar implications as the external entity expansion issue.

decompression bomb Decompression bombs (aka ZIP bomb) apply to all XML libraries that can parse compressed XML streams such as gzipped HTTP streams or LZMA-compressed files. For an attacker it can reduce the amount of transmitted data by three magnitudes or more.

The documentation for defusedxml on PyPI has further information about all known attack vectors with examples and references.

21.4.2 The `defusedxml` and `defusedexpat` Packages

defusedxml is a pure Python package with modified subclasses of all stdlib XML parsers that prevent any potentially malicious operation. Use of this package is recommended for any server code that parses untrusted XML data. The package also ships with example exploits and extended documentation on more XML exploits such as XPath injection.

defusedexpat provides a modified libexpat and a patched `pyexpat` module that have countermeasures against entity expansion DoS attacks. The `defusedexpat` module still allows a sane and configurable amount of entity expansions. The modifications may be included in some future release of Python, but will not be included in any bugfix releases of Python because they break backward compatibility.

21.5 API XML ElementTree

Código Fonte: Lib/xml/etree/ElementTree.py

O módulo *xml.etree.ElementTree* implementa uma API simples e eficiente para análise e criação de dados XML.

Alterado na versão 3.3: Esse módulo usará uma implementação rápida sempre que esta estiver disponível. O módulo `xml.etree.cElementTree` está obsoleto.

> **Aviso:** O módulo `xml.etree.cElementTree` não é seguro contra dados maliciosamente construídos. Se você precisa analisar dados não-confiáveis ou não-autenticados seja *XML vulnerabilities*.

21.5.1 Tutorial

Esse é um tutorial curto para usar *xml.etree.ElementTree* (ET na versão resumida). O objetivo é demonstrar alguns conceitos básicos e trechos de códigos do módulo.

XML tree e elementos

XML é um formato de dados estritamento hierárquico, e a maneira mais natural de representá-lo é como uma árvore. ET possui duas classes para esse propósito - *ElementTree* representa todo o documento XML como uma árvore e *Element* representa um único nó desta árvore. Interações com o documento inteiro (ler e escrever de/para arquivos) são frequentemente feitos em nível de *ElementTree*. Interações com um único elemento XML e seus subelementos são feitos a nível de *Element* .

Analisar XML

Nós utilizaremos o seguinte documento XML como exemplo de dados nessa seção:'

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank>68</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

Nós podemos importar esses dados lendo de um arquivo:

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

Ou diretamente de uma string:

```
root = ET.fromstring(country_data_as_string)
```

fromstring() obtém o XML de uma string e armazena em um *Element*, que será o elemento raiz dessa árvore. Outras funções de parsing podem criar um *ElementTree*. Cheque a documentação para se certificar sobre qual dado será retornado.

Assim como um *Element*, os elementos raiz tem uma tag e um dicionário de atributos:

```
>>> root.tag
'data'
>>> root.attrib
{}
```

Ele também tem nós filhos sobre os quais nós podemos iterar:

```
>>> for child in root:
...     print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

Nós filhos são os mais próximos, e nós podemos especificá-los por índices:

```
>>> root[0][1].text
'2008'
```

Nota: Not all elements of the XML input will end up as elements of the parsed tree. Currently, this module skips over any XML comments, processing instructions, and document type declarations in the input. Nevertheless, trees built using this module's API rather than parsing from XML text can have comments and processing instructions in them; they will be included when generating XML output. A document type declaration may be accessed by passing a custom *TreeBuilder* instance to the *XMLParser* constructor.

Pull API for non-blocking parsing

Most parsing functions provided by this module require the whole document to be read at once before returning any result. It is possible to use an *XMLParser* and feed data into it incrementally, but it is a push API that calls methods on a callback target, which is too low-level and inconvenient for most needs. Sometimes what the user really wants is to be able to parse XML incrementally, without blocking operations, while enjoying the convenience of fully constructed *Element* objects.

The most powerful tool for doing this is *XMLPullParser*. It does not require a blocking read to obtain the XML data, and is instead fed with data incrementally with *XMLPullParser.feed()* calls. To get the parsed XML elements, call *XMLPullParser.read_events()*. Here is an example:

```
>>> parser = ET.XMLPullParser(['start', 'end'])
>>> parser.feed('<mytag>sometext')
>>> list(parser.read_events())
[('start', <Element 'mytag' at 0x7fa66db2be58>)]
>>> parser.feed(' more text</mytag>')
>>> for event, elem in parser.read_events():
...     print(event)
...     print(elem.tag, 'text=', elem.text)
...
end
```

The obvious use case is applications that operate in a non-blocking fashion where the XML data is being received from a socket or read incrementally from some storage device. In such cases, blocking reads are unacceptable.

Because it's so flexible, *XMLPullParser* can be inconvenient to use for simpler use-cases. If you don't mind your application blocking on reading XML data but would still like to have incremental parsing capabilities, take a look at

iterparse(). It can be useful when you're reading a large XML document and don't want to hold it wholly in memory.

Finding interesting elements

Element has some useful methods that help iterate recursively over all the sub-tree below it (its children, their children, and so on). For example, *Element.iter()*:

```
>>> for neighbor in root.iter('neighbor'):
...     print(neighbor.attrib)
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

Element.findall() finds only elements with a tag which are direct children of the current element. *Element.find()* finds the *first* child with a particular tag, and *Element.text* accesses the element's text content. *Element.get()* accesses the element's attributes:

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

More sophisticated specification of which elements to look for is possible by using *XPath*.

Modifying an XML File

ElementTree provides a simple way to build XML documents and write them to files. The *ElementTree.write()* method serves this purpose.

Once created, an *Element* object may be manipulated by directly changing its fields (such as *Element.text*), adding and modifying attributes (*Element.set()* method), as well as adding new children (for example with *Element.append()*).

Let's say we want to add one to each country's rank, and add an `updated` attribute to the rank element:

```
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...
>>> tree.write('output.xml')
```

Our XML now looks like this:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
```

(continua na próxima página)

(continuação da página anterior)

```
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank updated="yes">69</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

We can remove elements using *Element.remove()*. Let's say we want to remove all countries with a rank higher than 50:

```
>>> for country in root.findall('country'):
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')
```

Our XML now looks like this:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
</data>
```

Building XML documents

The *SubElement()* function also provides a convenient way to create new sub-elements for a given element:

```
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>
```

Parsing XML with Namespaces

If the XML input has namespaces, tags and attributes with prefixes in the form `prefix:sometag` get expanded to `{uri}sometag` where the *prefix* is replaced by the full *URI*. Also, if there is a default namespace, that full URI gets prepended to all of the non-prefixed tags.

Here is an XML example that incorporates two namespaces, one with the prefix "fictional" and the other serving as the default namespace:

```
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
        xmlns="http://people.example.com">
    <actor>
        <name>John Cleese</name>
        <fictional:character>Lancelot</fictional:character>
        <fictional:character>Archie Leach</fictional:character>
    </actor>
    <actor>
        <name>Eric Idle</name>
        <fictional:character>Sir Robin</fictional:character>
        <fictional:character>Gunther</fictional:character>
        <fictional:character>Commander Clement</fictional:character>
    </actor>
</actors>
```

One way to search and explore this XML example is to manually add the URI to every tag or attribute in the xpath of a *find()* or *findall()*:

```
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor'):
    name = actor.find('{http://people.example.com}name')
    print(name.text)
    for char in actor.findall('{http://characters.example.com}character'):
        print(' |-->', char.text)
```

A better way to search the namespaced XML example is to create a dictionary with your own prefixes and use those in the search functions:

```
ns = {'real_person': 'http://people.example.com',
      'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print(name.text)
    for char in actor.findall('role:character', ns):
        print(' |-->', char.text)
```

These two approaches both output:

```
John Cleese
 |--> Lancelot
 |--> Archie Leach
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement
```

Additional resources

See http://effbot.org/zone/element-index.htm for tutorials and links to other docs.

21.5.2 XPath support

This module provides limited support for XPath expressions for locating elements in a tree. The goal is to support a small subset of the abbreviated syntax; a full XPath engine is outside the scope of the module.

Exemplo

Here's an example that demonstrates some of the XPath capabilities of the module. We'll be using the `countrydata` XML document from the *Parsing XML* section:

```
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the top-level
# elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall(".//year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall(".//*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of their parent
root.findall(".//neighbor[2]")
```

Supported XPath syntax

Sintaxe	Significado
`tag`	Selects all child elements with the given tag. For example, `spam` selects all child elements named `spam`, and `spam/egg` selects all grandchildren named `egg` in all children named `spam`.
`*`	Selects all child elements. For example, `*/egg` selects all grandchildren named `egg`.
`.`	Selects the current node. This is mostly useful at the beginning of the path, to indicate that it's a relative path.
`//`	Selects all subelements, on all levels beneath the current element. For example, `.//egg` selects all `egg` elements in the entire tree.
`..`	Selects the parent element. Returns `None` if the path attempts to reach the ancestors of the start element (the element `find` was called on).
`[@attrib]`	Selects all elements that have the given attribute.
`[@attrib='value']`	Selects all elements for which the given attribute has the given value. The value cannot contain quotes.
`[tag]`	Selects all elements that have a child named `tag`. Only immediate children are supported.
`[.='text']`	Selects all elements whose complete text content, including descendants, equals the given `text`. Novo na versão 3.7.
`[tag='text']`	Selects all elements that have a child named `tag` whose complete text content, including descendants, equals the given `text`.
`[position]`	Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression `last()` (for the last position), or a position relative to the last position (e.g. `last()-1`).

Predicates (expressions within square brackets) must be preceded by a tag name, an asterisk, or another predicate. `position` predicates must be preceded by a tag name.

21.5.3 Referência

Funções

xml.etree.ElementTree.**Comment**(*text=None*)
Comment element factory. This factory function creates a special element that will be serialized as an XML comment by the standard serializer. The comment string can be either a bytestring or a Unicode string. *text* is a string containing the comment string. Returns an element instance representing a comment.

Note that *XMLParser* skips over comments in the input instead of creating comment objects for them. An *ElementTree* will only contain comment nodes if they have been inserted into to the tree using one of the *Element* methods.

xml.etree.ElementTree.**dump**(*elem*)
Writes an element tree or element structure to sys.stdout. This function should be used for debugging only.

The exact output format is implementation dependent. In this version, it's written as an ordinary XML file.

elem is an element tree or an individual element.

xml.etree.ElementTree.**fromstring**(*text*, *parser=None*)
Parses an XML section from a string constant. Same as *XML()*. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns an *Element* instance.

xml.etree.ElementTree.**fromstringlist**(*sequence*, *parser=None*)
Parses an XML document from a sequence of string fragments. *sequence* is a list or other sequence containing XML data fragments. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `Element` instance.

Novo na versão 3.2.

xml.etree.ElementTree.**iselement**(*element*)
Check if an object appears to be a valid element object. *element* is an element instance. Return `True` if this is an element object.

xml.etree.ElementTree.**iterparse**(*source*, *events=None*, *parser=None*)
Parses an XML section into an element tree incrementally, and reports what's going on to the user. *source* is a filename or *file object* containing XML data. *events* is a sequence of events to report back. The supported events are the strings `"start"`, `"end"`, `"start-ns"` and `"end-ns"` (the "ns" events are used to get detailed namespace information). If *events* is omitted, only `"end"` events are reported. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. *parser* must be a subclass of `XMLParser` and can only use the default `TreeBuilder` as a target. Returns an *iterator* providing `(event, elem)` pairs.

Note that while `iterparse()` builds the tree incrementally, it issues blocking reads on *source* (or the file it names). As such, it's unsuitable for applications where blocking reads can't be made. For fully non-blocking parsing, see `XMLPullParser`.

Nota: `iterparse()` only guarantees that it has seen the ">" character of a starting tag when it emits a "start" event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.

If you need a fully populated element, look for "end" events instead.

Obsoleto desde a versão 3.4: The *parser* argument.

xml.etree.ElementTree.**parse**(*source*, *parser=None*)
Parses an XML section into an element tree. *source* is a filename or file object containing XML data. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `ElementTree` instance.

xml.etree.ElementTree.**ProcessingInstruction**(*target*, *text=None*)
PI element factory. This factory function creates a special element that will be serialized as an XML processing instruction. *target* is a string containing the PI target. *text* is a string containing the PI contents, if given. Returns an element instance, representing a processing instruction.

Note that `XMLParser` skips over processing instructions in the input instead of creating comment objects for them. An `ElementTree` will only contain processing instruction nodes if they have been inserted into to the tree using one of the `Element` methods.

xml.etree.ElementTree.**register_namespace**(*prefix*, *uri*)
Registers a namespace prefix. The registry is global, and any existing mapping for either the given prefix or the namespace URI will be removed. *prefix* is a namespace prefix. *uri* is a namespace uri. Tags and attributes in this namespace will be serialized with the given prefix, if at all possible.

Novo na versão 3.2.

xml.etree.ElementTree.**SubElement**(*parent*, *tag*, *attrib={}*, ***extra*)
Subelement factory. This function creates an element instance, and appends it to an existing element.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. *parent* is the parent element. *tag* is the subelement name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments. Returns an element instance.

xml.etree.ElementTree.**tostring**(*element*, *encoding="us-ascii"*, *method="xml"*, ***, *short_empty_elements=True*)

Generates a string representation of an XML element, including all subelements. *element* is an `Element` instance. *encoding*[1] is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). *short_empty_elements* has the same meaning as in `ElementTree.write()`. Returns an (optionally) encoded string containing the XML data.

Novo na versão 3.4: The *short_empty_elements* parameter.

xml.etree.ElementTree.**tostringlist**(*element*, *encoding="us-ascii"*, *method="xml"*, ***, *short_empty_elements=True*)

Generates a string representation of an XML element, including all subelements. *element* is an `Element` instance. *encoding*[1] is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). *short_empty_elements* has the same meaning as in `ElementTree.write()`. Returns a list of (optionally) encoded strings containing the XML data. It does not guarantee any specific sequence, except that `b"".join(tostringlist(element)) == tostring(element)`.

Novo na versão 3.2.

Novo na versão 3.4: The *short_empty_elements* parameter.

xml.etree.ElementTree.**XML**(*text*, *parser=None*)

Parses an XML section from a string constant. This function can be used to embed "XML literals" in Python code. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `Element` instance.

xml.etree.ElementTree.**XMLID**(*text*, *parser=None*)

Parses an XML section from a string constant, and also returns a dictionary which maps from element id:s to elements. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns a tuple containing an `Element` instance and a dictionary.

21.5.4 XInclude support

This module provides limited support for XInclude directives, via the `xml.etree.ElementInclude` helper module. This module can be used to insert subtrees and text strings into element trees, based on information in the tree.

Exemplo

Here's an example that demonstrates use of the XInclude module. To include an XML document in the current document, use the `{http://www.w3.org/2001/XInclude}include` element and set the **parse** attribute to `"xml"`, and use the **href** attribute to specify the document to include.

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="source.xml" parse="xml" />
</document>
```

By default, the **href** attribute is treated as a file name. You can use custom loaders to override this behaviour. Also note that the standard helper does not support XPointer syntax.

To process this file, load it as usual, and pass the root element to the `xml.etree.ElementTree` module:

[1] The encoding string included in XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

```
from xml.etree import ElementTree, ElementInclude

tree = ElementTree.parse("document.xml")
root = tree.getroot()

ElementInclude.include(root)
```

The ElementInclude module replaces the `{http://www.w3.org/2001/XInclude}include` element with the root element from the **source.xml** document. The result might look something like this:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <para>This is a paragraph.</para>
</document>
```

If the **parse** attribute is omitted, it defaults to "xml". The href attribute is required.

To include a text document, use the `{http://www.w3.org/2001/XInclude}include` element, and set the **parse** attribute to "text":

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) <xi:include href="year.txt" parse="text" />.
</document>
```

The result might look something like:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) 2003.
</document>
```

21.5.5 Referência

Funções

`xml.etree.ElementInclude.`**`default_loader`**(*href*, *parse*, *encoding=None*)

Default loader. This default loader reads an included resource from disk. *href* is a URL. *parse* is for parse mode either "xml" or "text". *encoding* is an optional text encoding. If not given, encoding is `utf-8`. Returns the expanded resource. If the parse mode is `"xml"`, this is an ElementTree instance. If the parse mode is "text", this is a Unicode string. If the loader fails, it can return None or raise an exception.

`xml.etree.ElementInclude.`**`include`**(*elem*, *loader=None*)

This function expands XInclude directives. *elem* is the root element. *loader* is an optional resource loader. If omitted, it defaults to *`default_loader()`*. If given, it should be a callable that implements the same interface as *`default_loader()`*. Returns the expanded resource. If the parse mode is `"xml"`, this is an ElementTree instance. If the parse mode is "text", this is a Unicode string. If the loader fails, it can return None or raise an exception.

Element Objects

class xml.etree.ElementTree.**Element**(*tag*, *attrib={}*, ***extra*)
Element class. This class defines the Element interface, and provides a reference implementation of this interface.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. *tag* is the element name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments.

tag
A string identifying what kind of data this element represents (the element type, in other words).

text
tail
These attributes can be used to hold additional data associated with the element. Their values are usually strings but may be any application-specific object. If the element is created from an XML file, the *text* attribute holds either the text between the element's start tag and its first child or end tag, or `None`, and the *tail* attribute holds either the text between the element's end tag and the next tag, or `None`. For the XML data

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

the *a* element has `None` for both *text* and *tail* attributes, the *b* element has *text* `"1"` and *tail* `"4"`, the *c* element has *text* `"2"` and *tail* `None`, and the *d* element has *text* `None` and *tail* `"3"`.

To collect the inner text of an element, see *itertext()*, for example `"".join(element.itertext())`.

Applications may store arbitrary objects in these attributes.

attrib
A dictionary containing the element's attributes. Note that while the *attrib* value is always a real mutable Python dictionary, an ElementTree implementation may choose to use another internal representation, and create the dictionary only if someone asks for it. To take advantage of such implementations, use the dictionary methods below whenever possible.

The following dictionary-like methods work on the element attributes.

clear()
Resets an element. This function removes all subelements, clears all attributes, and sets the text and tail attributes to `None`.

get(*key*, *default=None*)
Gets the element attribute named *key*.

Returns the attribute value, or *default* if the attribute was not found.

items()
Returns the element attributes as a sequence of (name, value) pairs. The attributes are returned in an arbitrary order.

keys()
Returns the elements attribute names as a list. The names are returned in an arbitrary order.

set(*key*, *value*)
Set the attribute *key* on the element to *value*.

The following methods work on the element's children (subelements).

append(*subelement*)
Adds the element *subelement* to the end of this element's internal list of subelements. Raises *TypeError* if *subelement* is not an *Element*.

extend(*subelements*)
Appends *subelements* from a sequence object with zero or more elements. Raises *TypeError* if a subelement is not an *Element*.

Novo na versão 3.2.

find(*match*, *namespaces=None*)
Finds the first subelement matching *match*. *match* may be a tag name or a *path*. Returns an element instance or `None`. *namespaces* is an optional mapping from namespace prefix to full name.

findall(*match*, *namespaces=None*)
Finds all matching subelements, by tag name or *path*. Returns a list containing all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name.

findtext(*match*, *default=None*, *namespaces=None*)
Finds text for the first subelement matching *match*. *match* may be a tag name or a *path*. Returns the text content of the first matching element, or *default* if no element was found. Note that if the matching element has no text content an empty string is returned. *namespaces* is an optional mapping from namespace prefix to full name.

getchildren()
Obsoleto desde a versão 3.2: Use `list(elem)` or iteration.

getiterator(*tag=None*)
Obsoleto desde a versão 3.2: Use method *Element.iter()* instead.

insert(*index*, *subelement*)
Inserts *subelement* at the given position in this element. Raises *TypeError* if *subelement* is not an *Element*.

iter(*tag=None*)
Creates a tree *iterator* with the current element as the root. The iterator iterates over this element and all elements below it, in document (depth first) order. If *tag* is not `None` or `'*'`, only elements whose tag equals *tag* are returned from the iterator. If the tree structure is modified during iteration, the result is undefined.

Novo na versão 3.2.

iterfind(*match*, *namespaces=None*)
Finds all matching subelements, by tag name or *path*. Returns an iterable yielding all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name.

Novo na versão 3.2.

itertext()
Creates a text iterator. The iterator loops over this element and all subelements, in document order, and returns all inner text.

Novo na versão 3.2.

makeelement(*tag*, *attrib*)
Creates a new element object of the same type as this element. Do not call this method, use the *SubElement()* factory function instead.

remove(*subelement*)
Removes *subelement* from the element. Unlike the find* methods this method compares elements based on the instance identity, not on tag value or contents.

Element objects also support the following sequence type methods for working with subelements: `__delitem__()`, `__getitem__()`, `__setitem__()`, `__len__()`.

Caution: Elements with no subelements will test as `False`. This behavior will change in future versions. Use specific `len(elem)` or `elem is None` test instead.

```
element = root.find('foo')

if not element:  # careful!
    print("element not found, or element has no subelements")

if element is None:
    print("element not found")
```

ElementTree Objects

class xml.etree.ElementTree.**ElementTree**(*element=None*, *file=None*)

ElementTree wrapper class. This class represents an entire element hierarchy, and adds some extra support for serialization to and from standard XML.

element is the root element. The tree is initialized with the contents of the XML *file* if given.

_setroot(*element*)

Replaces the root element for this tree. This discards the current contents of the tree, and replaces it with the given element. Use with care. *element* is an element instance.

find(*match*, *namespaces=None*)

Same as *Element.find()*, starting at the root of the tree.

findall(*match*, *namespaces=None*)

Same as *Element.findall()*, starting at the root of the tree.

findtext(*match*, *default=None*, *namespaces=None*)

Same as *Element.findtext()*, starting at the root of the tree.

getiterator(*tag=None*)

Obsoleto desde a versão 3.2: Use method *ElementTree.iter()* instead.

getroot()

Returns the root element for this tree.

iter(*tag=None*)

Creates and returns a tree iterator for the root element. The iterator loops over all elements in this tree, in section order. *tag* is the tag to look for (default is to return all elements).

iterfind(*match*, *namespaces=None*)

Same as *Element.iterfind()*, starting at the root of the tree.

Novo na versão 3.2.

parse(*source*, *parser=None*)

Loads an external XML section into this element tree. *source* is a file name or *file object*. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. Returns the section root element.

write(*file*, *encoding="us-ascii"*, *xml_declaration=None*, *default_namespace=None*, *method="xml"*, *, *short_empty_elements=True*)

Writes the element tree to a file, as XML. *file* is a file name, or a *file object* opened for writing. *encoding*[1] is the output encoding (default is US-ASCII). *xml_declaration* controls if an XML declaration should be added to the file. Use `False` for never, `True` for always, `None` for only if not US-ASCII or UTF-8 or Unicode (default is `None`). *default_namespace* sets the default XML namespace (for "xmlns"). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). The keyword-only *short_empty_elements* parameter controls the formatting of elements that contain no content. If `True` (the default), they are emitted as a single self-closed tag, otherwise they are emitted as a pair of start/end tags.

The output is either a string (*str*) or binary (*bytes*). This is controlled by the *encoding* argument. If *encoding* is `"unicode"`, the output is a string; otherwise, it's binary. Note that this may conflict with the type of *file* if it's an open *file object*; make sure you do not try to write a string to a binary stream and vice versa.

Novo na versão 3.4: The *short_empty_elements* parameter.

This is the XML file that is going to be manipulated:

```
<html>
    <head>
        <title>Example page</title>
    </head>
    <body>
        <p>Moved to <a href="http://example.org/">example.org</a>
        or <a href="http://example.com/">example.com</a>.</p>
    </body>
</html>
```

Example of changing the attribute "target" of every link in first paragraph:

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")     # Finds first occurrence of tag p in body
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))   # Returns list of all links
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:             # Iterates through all found links
...     i.attrib["target"] = "blank"
>>> tree.write("output.xhtml")
```

QName Objects

class xml.etree.ElementTree.**QName**(*text_or_uri*, *tag=None*)

QName wrapper. This can be used to wrap a QName attribute value, in order to get proper namespace handling on output. *text_or_uri* is a string containing the QName value, in the form {uri}local, or, if the tag argument is given, the URI part of a QName. If *tag* is given, the first argument is interpreted as a URI, and this argument is interpreted as a local name. *QName* instances are opaque.

TreeBuilder Objects

class xml.etree.ElementTree.**TreeBuilder**(*element_factory=None*)

Generic element structure builder. This builder converts a sequence of start, data, and end method calls to a well-formed element structure. You can use this class to build an element structure using a custom XML parser, or a parser for some other XML-like format. *element_factory*, when given, must be a callable accepting two positional arguments: a tag and a dict of attributes. It is expected to return a new element instance.

close()

Flushes the builder buffers, and returns the toplevel document element. Returns an *Element* instance.

data(*data*)

Adds text to the current element. *data* is a string. This should be either a bytestring, or a Unicode string.

end(*tag*)
Closes the current element. *tag* is the element name. Returns the closed element.

start(*tag*, *attrs*)
Opens a new element. *tag* is the element name. *attrs* is a dictionary containing element attributes. Returns the opened element.

In addition, a custom `TreeBuilder` object can provide the following method:

doctype(*name*, *pubid*, *system*)
Handles a doctype declaration. *name* is the doctype name. *pubid* is the public identifier. *system* is the system identifier. This method does not exist on the default `TreeBuilder` class.

Novo na versão 3.2.

Objetos XMLParser

class xml.etree.ElementTree.**XMLParser**(*html=0*, *target=None*, *encoding=None*)
This class is the low-level building block of the module. It uses `xml.parsers.expat` for efficient, event-based parsing of XML. It can be fed XML data incrementally with the `feed()` method, and parsing events are translated to a push API - by invoking callbacks on the *target* object. If *target* is omitted, the standard `TreeBuilder` is used. The *html* argument was historically used for backwards compatibility and is now deprecated. If *encoding*[1] is given, the value overrides the encoding specified in the XML file.

Obsoleto desde a versão 3.4: The *html* argument. The remaining arguments should be passed via keyword to prepare for the removal of the *html* argument.

close()
Finishes feeding data to the parser. Returns the result of calling the `close()` method of the *target* passed during construction; by default, this is the toplevel document element.

doctype(*name*, *pubid*, *system*)
Obsoleto desde a versão 3.2: Define the `TreeBuilder.doctype()` method on a custom TreeBuilder target.

feed(*data*)
Feeds data to the parser. *data* is encoded data.

`XMLParser.feed()` calls *target*'s `start(tag, attrs_dict)` method for each opening tag, its `end(tag)` method for each closing tag, and data is processed by method `data(data)`. `XMLParser.close()` calls *target*'s method `close()`. `XMLParser` can be used not only for building a tree structure. This is an example of counting the maximum depth of an XML file:

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                     # The target object of the parser
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):   # Called for each opening tag.
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):             # Called for each closing tag.
...         self.depth -= 1
...     def data(self, data):
...         pass            # We do not need to do anything with data.
...     def close(self):    # Called when all data has been parsed.
...         return self.maxDepth
...
```

(continua na próxima página)

(continuação da página anterior)

```
>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
...   <b>
...   </b>
...   <b>
...     <c>
...       <d>
...       </d>
...     </c>
...   </b>
... </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()
4
```

XMLPullParser Objects

class xml.etree.ElementTree.**XMLPullParser**(*events=None*)

A pull parser suitable for non-blocking applications. Its input-side API is similar to that of *XMLParser*, but instead of pushing calls to a callback target, *XMLPullParser* collects an internal list of parsing events and lets the user read from it. *events* is a sequence of events to report back. The supported events are the strings `"start"`, `"end"`, `"start-ns"` and `"end-ns"` (the "ns" events are used to get detailed namespace information). If *events* is omitted, only `"end"` events are reported.

feed(*data*)

Feed the given bytes data to the parser.

close()

Signal the parser that the data stream is terminated. Unlike *XMLParser.close()*, this method always returns *None*. Any events not yet retrieved when the parser is closed can still be read with *read_events()*.

read_events()

Return an iterator over the events which have been encountered in the data fed to the parser. The iterator yields `(event, elem)` pairs, where *event* is a string representing the type of event (e.g. `"end"`) and *elem* is the encountered *Element* object.

Events provided in a previous call to *read_events()* will not be yielded again. Events are consumed from the internal queue only when they are retrieved from the iterator, so multiple readers iterating in parallel over iterators obtained from *read_events()* will have unpredictable results.

Nota: *XMLPullParser* only guarantees that it has seen the ">" character of a starting tag when it emits a "start" event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.

If you need a fully populated element, look for "end" events instead.

Novo na versão 3.4.

Exceções

class xml.etree.ElementTree.**ParseError**
XML parse error, raised by the various parsing methods in this module when parsing fails. The string representation of an instance of this exception will contain a user-friendly error message. In addition, it will have the following attributes available:

code
A numeric error code from the expat parser. See the documentation of *xml.parsers.expat* for the list of error codes and their meanings.

position
A tuple of *line*, *column* numbers, specifying where the error occurred.

21.6 xml.dom — The Document Object Model API

Código Fonte: Lib/xml/dom/__init__.py

The Document Object Model, or "DOM," is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents. A DOM implementation presents an XML document as a tree structure, or allows client code to build such a structure from scratch. It then gives access to the structure through a set of objects which provided well-known interfaces.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another. If you are looking at a text node, you have no access to a containing element. When you write a SAX application, you need to keep track of your program's position in the document somewhere in your own code. SAX does not do it for you. Also, if you need to look ahead in the XML document, you are just out of luck.

Some applications are simply impossible in an event driven model with no access to a tree. Of course you could build some sort of tree yourself in SAX events, but the DOM allows you to avoid writing that code. The DOM is a standard tree representation for XML data.

The Document Object Model is being defined by the W3C in stages, or "levels" in their terminology. The Python mapping of the API is substantially based on the DOM Level 2 recommendation.

DOM applications typically start by parsing some XML into a DOM. How this is accomplished is not covered at all by DOM Level 1, and Level 2 provides only limited improvements: There is a `DOMImplementation` object class which provides access to `Document` creation methods, but no way to access an XML reader/parser/Document builder in an implementation-independent way. There is also no well-defined way to access these methods without an existing `Document` object. In Python, each DOM implementation will provide a function *getDOMImplementation()*. DOM Level 3 adds a Load/Store specification, which defines an interface to the reader, but this is not yet available in the Python standard library.

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification; this portion of the reference manual describes the interpretation of the specification in Python.

The specification provided by the W3C defines the DOM API for Java, ECMAScript, and OMG IDL. The Python mapping defined here is based in large part on the IDL version of the specification, but strict compliance is not required (though implementations are free to support the strict mapping from IDL). See section *Conformance* for a detailed discussion of mapping requirements.

Ver também:

Document Object Model (DOM) Level 2 Specification The W3C recommendation upon which the Python DOM API is based.

Document Object Model (DOM) Level 1 Specification The W3C recommendation for the DOM supported by *xml.dom.minidom*.

Python Language Mapping Specification This specifies the mapping from OMG IDL to Python.

21.6.1 Conteúdo do Módulo

The *xml.dom* contains the following functions:

xml.dom.**registerDOMImplementation**(*name*, *factory*)
Register the *factory* function with the name *name*. The factory function should return an object which implements the DOMImplementation interface. The factory function can return the same object every time, or a new one for each call, as appropriate for the specific implementation (e.g. if that implementation supports some customization).

xml.dom.**getDOMImplementation**(*name=None*, *features=()*)
Return a suitable DOM implementation. The *name* is either well-known, the module name of a DOM implementation, or None. If it is not None, imports the corresponding module and returns a DOMImplementation object if the import succeeds. If no name is given, and if the environment variable PYTHON_DOM is set, this variable is used to find the implementation.

If name is not given, this examines the available implementations to find one with the required feature set. If no implementation can be found, raise an *ImportError*. The features list must be a sequence of (feature, version) pairs which are passed to the hasFeature() method on available DOMImplementation objects.

Some convenience constants are also provided:

xml.dom.**EMPTY_NAMESPACE**
The value used to indicate that no namespace is associated with a node in the DOM. This is typically found as the namespaceURI of a node, or used as the *namespaceURI* parameter to a namespaces-specific method.

xml.dom.**XML_NAMESPACE**
The namespace URI associated with the reserved prefix xml, as defined by Namespaces in XML (section 4).

xml.dom.**XMLNS_NAMESPACE**
The namespace URI for namespace declarations, as defined by Document Object Model (DOM) Level 2 Core Specification (section 1.1.8).

xml.dom.**XHTML_NAMESPACE**
The URI of the XHTML namespace as defined by XHTML 1.0: The Extensible HyperText Markup Language (section 3.1.1).

In addition, *xml.dom* contains a base Node class and the DOM exception classes. The Node class provided by this module does not implement any of the methods or attributes defined by the DOM specification; concrete DOM implementations must provide those. The Node class provided as part of this module does provide the constants used for the nodeType attribute on concrete Node objects; they are located within the class rather than at the module level to conform with the DOM specifications.

21.6.2 Objects in the DOM

The definitive documentation for the DOM is the DOM specification from the W3C.

Note that DOM attributes may also be manipulated as nodes instead of as simple strings. It is fairly rare that you must do this, however, so this usage is not yet documented.

Interface	Seção	Purpose
`DOMImplementation`	*DOMImplementation Objects*	Interface to the underlying implementation.
`Node`	*Objetos Node*	Base interface for most objects in a document.
`NodeList`	*Objetos NodeList*	Interface for a sequence of nodes.
`DocumentType`	*DocumentType Objects*	Information about the declarations needed to process a document.
`Document`	*Document Objects*	Object which represents an entire document.
`Element`	*Element Objects*	Element nodes in the document hierarchy.
`Attr`	*Attr Objects*	Attribute value nodes on element nodes.
`Comment`	*Comment Objects*	Representation of comments in the source document.
`Text`	*Text and CDATASection Objects*	Nodes containing textual content from the document.
`ProcessingInstruction`	*Objetos ProcessingInstruction*	Processing instruction representation.

An additional section describes the exceptions defined for working with the DOM in Python.

DOMImplementation Objects

The `DOMImplementation` interface provides a way for applications to determine the availability of particular features in the DOM they are using. DOM Level 2 added the ability to create new `Document` and `DocumentType` objects using the `DOMImplementation` as well.

`DOMImplementation.`**`hasFeature`**(*feature*, *version*)
Return `True` if the feature identified by the pair of strings *feature* and *version* is implemented.

`DOMImplementation.`**`createDocument`**(*namespaceUri*, *qualifiedName*, *doctype*)
Return a new `Document` object (the root of the DOM), with a child `Element` object having the given *namespaceUri* and *qualifiedName*. The *doctype* must be a `DocumentType` object created by `createDocumentType()`, or `None`. In the Python DOM API, the first two arguments can also be `None` in order to indicate that no `Element` child is to be created.

`DOMImplementation.`**`createDocumentType`**(*qualifiedName*, *publicId*, *systemId*)
Return a new `DocumentType` object that encapsulates the given *qualifiedName*, *publicId*, and *systemId* strings, representing the information contained in an XML document type declaration.

Objetos Node

All of the components of an XML document are subclasses of `Node`.

`Node`.**nodeType**
An integer representing the node type. Symbolic constants for the types are on the `Node` object: `ELEMENT_NODE`, `ATTRIBUTE_NODE`, `TEXT_NODE`, `CDATA_SECTION_NODE`, `ENTITY_NODE`, `PROCESSING_INSTRUCTION_NODE`, `COMMENT_NODE`, `DOCUMENT_NODE`, `DOCUMENT_TYPE_NODE`, `NOTATION_NODE`. This is a read-only attribute.

`Node`.**parentNode**
The parent of the current node, or `None` for the document node. The value is always a `Node` object or `None`. For `Element` nodes, this will be the parent element, except for the root element, in which case it will be the `Document` object. For `Attr` nodes, this is always `None`. This is a read-only attribute.

`Node`.**attributes**
A `NamedNodeMap` of attribute objects. Only elements have actual values for this; others provide `None` for this attribute. This is a read-only attribute.

`Node`.**previousSibling**
The node that immediately precedes this one with the same parent. For instance the element with an end-tag that comes just before the *self* element's start-tag. Of course, XML documents are made up of more than just elements so the previous sibling could be text, a comment, or something else. If this node is the first child of the parent, this attribute will be `None`. This is a read-only attribute.

`Node`.**nextSibling**
The node that immediately follows this one with the same parent. See also *`previousSibling`*. If this is the last child of the parent, this attribute will be `None`. This is a read-only attribute.

`Node`.**childNodes**
A list of nodes contained within this node. This is a read-only attribute.

`Node`.**firstChild**
The first child of the node, if there are any, or `None`. This is a read-only attribute.

`Node`.**lastChild**
The last child of the node, if there are any, or `None`. This is a read-only attribute.

`Node`.**localName**
The part of the `tagName` following the colon if there is one, else the entire `tagName`. The value is a string.

`Node`.**prefix**
The part of the `tagName` preceding the colon if there is one, else the empty string. The value is a string, or `None`.

`Node`.**namespaceURI**
The namespace associated with the element name. This will be a string or `None`. This is a read-only attribute.

`Node`.**nodeName**
This has a different meaning for each node type; see the DOM specification for details. You can always get the information you would get here from another property such as the `tagName` property for elements or the `name` property for attributes. For all node types, the value of this attribute will be either a string or `None`. This is a read-only attribute.

`Node`.**nodeValue**
This has a different meaning for each node type; see the DOM specification for details. The situation is similar to that with *`nodeName`*. The value is a string or `None`.

`Node`.**hasAttributes**`()`
Return `True` if the node has any attributes.

Node.**hasChildNodes**()
Return True if the node has any child nodes.

Node.**isSameNode**(*other*)
Return True if *other* refers to the same node as this node. This is especially useful for DOM implementations which use any sort of proxy architecture (because more than one object can refer to the same node).

Nota: This is based on a proposed DOM Level 3 API which is still in the "working draft" stage, but this particular interface appears uncontroversial. Changes from the W3C will not necessarily affect this method in the Python DOM interface (though any new W3C API for this would also be supported).

Node.**appendChild**(*newChild*)
Add a new child node to this node at the end of the list of children, returning *newChild*. If the node was already in the tree, it is removed first.

Node.**insertBefore**(*newChild*, *refChild*)
Insert a new child node before an existing child. It must be the case that *refChild* is a child of this node; if not, *ValueError* is raised. *newChild* is returned. If *refChild* is None, it inserts *newChild* at the end of the children's list.

Node.**removeChild**(*oldChild*)
Remove a child node. *oldChild* must be a child of this node; if not, *ValueError* is raised. *oldChild* is returned on success. If *oldChild* will not be used further, its unlink() method should be called.

Node.**replaceChild**(*newChild*, *oldChild*)
Replace an existing node with a new node. It must be the case that *oldChild* is a child of this node; if not, *ValueError* is raised.

Node.**normalize**()
Join adjacent text nodes so that all stretches of text are stored as single Text instances. This simplifies processing text from a DOM tree for many applications.

Node.**cloneNode**(*deep*)
Clone this node. Setting *deep* means to clone all child nodes as well. This returns the clone.

Objetos NodeList

A NodeList represents a sequence of nodes. These objects are used in two ways in the DOM Core recommendation: an Element object provides one as its list of child nodes, and the getElementsByTagName() and getElementsByTagNameNS() methods of Node return objects with this interface to represent query results.

The DOM Level 2 recommendation defines one method and one attribute for these objects:

NodeList.**item**(*i*)
Return the *i*'th item from the sequence, if there is one, or None. The index *i* is not allowed to be less than zero or greater than or equal to the length of the sequence.

NodeList.**length**
The number of nodes in the sequence.

In addition, the Python DOM interface requires that some additional support is provided to allow NodeList objects to be used as Python sequences. All NodeList implementations must include support for __len__() and __getitem__(); this allows iteration over the NodeList in for statements and proper support for the *len()* built-in function.

If a DOM implementation supports modification of the document, the NodeList implementation must also support the __setitem__() and __delitem__() methods.

DocumentType Objects

Information about the notations and entities declared by a document (including the external subset if the parser uses it and can provide the information) is available from a `DocumentType` object. The `DocumentType` for a document is available from the `Document` object's `doctype` attribute; if there is no `DOCTYPE` declaration for the document, the document's `doctype` attribute will be set to `None` instead of an instance of this interface.

`DocumentType` is a specialization of `Node`, and adds the following attributes:

`DocumentType.`**`publicId`**
The public identifier for the external subset of the document type definition. This will be a string or `None`.

`DocumentType.`**`systemId`**
The system identifier for the external subset of the document type definition. This will be a URI as a string, or `None`.

`DocumentType.`**`internalSubset`**
A string giving the complete internal subset from the document. This does not include the brackets which enclose the subset. If the document has no internal subset, this should be `None`.

`DocumentType.`**`name`**
The name of the root element as given in the `DOCTYPE` declaration, if present.

`DocumentType.`**`entities`**
This is a `NamedNodeMap` giving the definitions of external entities. For entity names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be `None` if the information is not provided by the parser, or if no entities are defined.

`DocumentType.`**`notations`**
This is a `NamedNodeMap` giving the definitions of notations. For notation names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be `None` if the information is not provided by the parser, or if no notations are defined.

Document Objects

A `Document` represents an entire XML document, including its constituent elements, attributes, processing instructions, comments etc. Remember that it inherits properties from `Node`.

`Document.`**`documentElement`**
The one and only root element of the document.

`Document.`**`createElement`**(*tagName*)
Create and return a new element node. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as `insertBefore()` or `appendChild()`.

`Document.`**`createElementNS`**(*namespaceURI*, *tagName*)
Create and return a new element with a namespace. The *tagName* may have a prefix. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as `insertBefore()` or `appendChild()`.

`Document.`**`createTextNode`**(*data*)
Create and return a text node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

`Document.`**`createComment`**(*data*)
Create and return a comment node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

Document.**createProcessingInstruction**(*target*, *data*)
Create and return a processing instruction node containing the *target* and *data* passed as parameters. As with the other creation methods, this one does not insert the node into the tree.

Document.**createAttribute**(*name*)
Create and return an attribute node. This method does not associate the attribute node with any particular element. You must use `setAttributeNode()` on the appropriate `Element` object to use the newly created attribute instance.

Document.**createAttributeNS**(*namespaceURI*, *qualifiedName*)
Create and return an attribute node with a namespace. The *tagName* may have a prefix. This method does not associate the attribute node with any particular element. You must use `setAttributeNode()` on the appropriate `Element` object to use the newly created attribute instance.

Document.**getElementsByTagName**(*tagName*)
Search for all descendants (direct children, children's children, etc.) with a particular element type name.

Document.**getElementsByTagNameNS**(*namespaceURI*, *localName*)
Search for all descendants (direct children, children's children, etc.) with a particular namespace URI and localname. The localname is the part of the namespace after the prefix.

Element Objects

`Element` is a subclass of `Node`, so inherits all the attributes of that class.

Element.**tagName**
The element type name. In a namespace-using document it may have colons in it. The value is a string.

Element.**getElementsByTagName**(*tagName*)
Same as equivalent method in the `Document` class.

Element.**getElementsByTagNameNS**(*namespaceURI*, *localName*)
Same as equivalent method in the `Document` class.

Element.**hasAttribute**(*name*)
Return `True` if the element has an attribute named by *name*.

Element.**hasAttributeNS**(*namespaceURI*, *localName*)
Return `True` if the element has an attribute named by *namespaceURI* and *localName*.

Element.**getAttribute**(*name*)
Return the value of the attribute named by *name* as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

Element.**getAttributeNode**(*attrname*)
Return the `Attr` node for the attribute named by *attrname*.

Element.**getAttributeNS**(*namespaceURI*, *localName*)
Return the value of the attribute named by *namespaceURI* and *localName* as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

Element.**getAttributeNodeNS**(*namespaceURI*, *localName*)
Return an attribute value as a node, given a *namespaceURI* and *localName*.

Element.**removeAttribute**(*name*)
Remove an attribute by name. If there is no matching attribute, a *NotFoundErr* is raised.

Element.**removeAttributeNode**(*oldAttr*)
Remove and return *oldAttr* from the attribute list, if present. If *oldAttr* is not present, *NotFoundErr* is raised.

Element.**removeAttributeNS**(*namespaceURI*, *localName*)
Remove an attribute by name. Note that it uses a localName, not a qname. No exception is raised if there is no matching attribute.

Element.**setAttribute**(*name*, *value*)
Set an attribute value from a string.

Element.**setAttributeNode**(*newAttr*)
Add a new attribute node to the element, replacing an existing attribute if necessary if the `name` attribute matches. If a replacement occurs, the old attribute node will be returned. If *newAttr* is already in use, *InuseAttributeErr* will be raised.

Element.**setAttributeNodeNS**(*newAttr*)
Add a new attribute node to the element, replacing an existing attribute if necessary if the `namespaceURI` and `localName` attributes match. If a replacement occurs, the old attribute node will be returned. If *newAttr* is already in use, *InuseAttributeErr* will be raised.

Element.**setAttributeNS**(*namespaceURI*, *qname*, *value*)
Set an attribute value from a string, given a *namespaceURI* and a *qname*. Note that a qname is the whole attribute name. This is different than above.

Attr Objects

`Attr` inherits from `Node`, so inherits all its attributes.

Attr.**name**
The attribute name. In a namespace-using document it may include a colon.

Attr.**localName**
The part of the name following the colon if there is one, else the entire name. This is a read-only attribute.

Attr.**prefix**
The part of the name preceding the colon if there is one, else the empty string.

Attr.**value**
The text value of the attribute. This is a synonym for the `nodeValue` attribute.

NamedNodeMap Objects

`NamedNodeMap` does *not* inherit from `Node`.

NamedNodeMap.**length**
The length of the attribute list.

NamedNodeMap.**item**(*index*)
Return an attribute with a particular index. The order you get the attributes in is arbitrary but will be consistent for the life of a DOM. Each item is an attribute node. Get its value with the `value` attribute.

There are also experimental methods that give this class more mapping behavior. You can use them or you can use the standardized `getAttribute*()` family of methods on the `Element` objects.

Comment Objects

`Comment` represents a comment in the XML document. It is a subclass of `Node`, but cannot have child nodes.

`Comment`**.data**
The content of the comment as a string. The attribute contains all characters between the leading `<!--` and trailing `-->`, but does not include them.

Text and CDATASection Objects

The `Text` interface represents text in the XML document. If the parser and DOM implementation support the DOM's XML extension, portions of the text enclosed in CDATA marked sections are stored in `CDATASection` objects. These two interfaces are identical, but provide different values for the `nodeType` attribute.

These interfaces extend the `Node` interface. They cannot have child nodes.

`Text`**.data**
The content of the text node as a string.

Nota: The use of a `CDATASection` node does not indicate that the node represents a complete CDATA marked section, only that the content of the node was part of a CDATA section. A single CDATA section may be represented by more than one node in the document tree. There is no way to determine whether two adjacent `CDATASection` nodes represent different CDATA marked sections.

Objetos ProcessingInstruction

Represents a processing instruction in the XML document; this inherits from the `Node` interface and cannot have child nodes.

`ProcessingInstruction`**.target**
The content of the processing instruction up to the first whitespace character. This is a read-only attribute.

`ProcessingInstruction`**.data**
The content of the processing instruction following the first whitespace character.

Exceções

The DOM Level 2 recommendation defines a single exception, *`DOMException`*, and a number of constants that allow applications to determine what sort of error occurred. *`DOMException`* instances carry a *`code`* attribute that provides the appropriate value for the specific exception.

The Python DOM interface provides the constants, but also expands the set of exceptions so that a specific exception exists for each of the exception codes defined by the DOM. The implementations must raise the appropriate specific exception, each of which carries the appropriate value for the *`code`* attribute.

exception `xml.dom`**.DOMException**
Base exception class used for all specific DOM exceptions. This exception class cannot be directly instantiated.

exception `xml.dom`**.DomstringSizeErr**
Raised when a specified range of text does not fit into a string. This is not known to be used in the Python DOM implementations, but may be received from DOM implementations not written in Python.

exception `xml.dom`**.HierarchyRequestErr**
Raised when an attempt is made to insert a node where the node type is not allowed.

exception xml.dom.**IndexSizeErr**
Raised when an index or size parameter to a method is negative or exceeds the allowed values.

exception xml.dom.**InuseAttributeErr**
Raised when an attempt is made to insert an `Attr` node that is already present elsewhere in the document.

exception xml.dom.**InvalidAccessErr**
Raised if a parameter or an operation is not supported on the underlying object.

exception xml.dom.**InvalidCharacterErr**
This exception is raised when a string parameter contains a character that is not permitted in the context it's being used in by the XML 1.0 recommendation. For example, attempting to create an `Element` node with a space in the element type name will cause this error to be raised.

exception xml.dom.**InvalidModificationErr**
Raised when an attempt is made to modify the type of a node.

exception xml.dom.**InvalidStateErr**
Raised when an attempt is made to use an object that is not defined or is no longer usable.

exception xml.dom.**NamespaceErr**
If an attempt is made to change any object in a way that is not permitted with regard to the Namespaces in XML recommendation, this exception is raised.

exception xml.dom.**NotFoundErr**
Exception when a node does not exist in the referenced context. For example, `NamedNodeMap.removeNamedItem()` will raise this if the node passed in does not exist in the map.

exception xml.dom.**NotSupportedErr**
Raised when the implementation does not support the requested type of object or operation.

exception xml.dom.**NoDataAllowedErr**
This is raised if data is specified for a node which does not support data.

exception xml.dom.**NoModificationAllowedErr**
Raised on attempts to modify an object where modifications are not allowed (such as for read-only nodes).

exception xml.dom.**SyntaxErr**
Raised when an invalid or illegal string is specified.

exception xml.dom.**WrongDocumentErr**
Raised when a node is inserted in a different document than it currently belongs to, and the implementation does not support migrating the node from one document to the other.

The exception codes defined in the DOM recommendation map to the exceptions described above according to this table:

Constante	Exception
DOMSTRING_SIZE_ERR	*DomstringSizeErr*
HIERARCHY_REQUEST_ERR	*HierarchyRequestErr*
INDEX_SIZE_ERR	*IndexSizeErr*
INUSE_ATTRIBUTE_ERR	*InuseAttributeErr*
INVALID_ACCESS_ERR	*InvalidAccessErr*
INVALID_CHARACTER_ERR	*InvalidCharacterErr*
INVALID_MODIFICATION_ERR	*InvalidModificationErr*
INVALID_STATE_ERR	*InvalidStateErr*
NAMESPACE_ERR	*NamespaceErr*
NOT_FOUND_ERR	*NotFoundErr*
NOT_SUPPORTED_ERR	*NotSupportedErr*
NO_DATA_ALLOWED_ERR	*NoDataAllowedErr*
NO_MODIFICATION_ALLOWED_ERR	*NoModificationAllowedErr*
SYNTAX_ERR	*SyntaxErr*
WRONG_DOCUMENT_ERR	*WrongDocumentErr*

21.6.3 Conformance

This section describes the conformance requirements and relationships between the Python DOM API, the W3C DOM recommendations, and the OMG IDL mapping for Python.

Type Mapping

The IDL types used in the DOM specification are mapped to Python types according to the following table.

Tipo IDL	Tipo em Python
`boolean`	`bool` or `int`
`int`	`int`
`long int`	`int`
`unsigned int`	`int`
`DOMString`	`str` or `bytes`
`null`	`None`

Accessor Methods

The mapping from OMG IDL to Python defines accessor functions for IDL `attribute` declarations in much the way the Java mapping does. Mapping the IDL declarations

```
readonly attribute string someValue;
         attribute string anotherValue;
```

yields three accessor functions: a "get" method for `someValue` (`_get_someValue()`), and "get" and "set" methods for `anotherValue` (`_get_anotherValue()` and `_set_anotherValue()`). The mapping, in particular, does not require that the IDL attributes are accessible as normal Python attributes: `object.someValue` is *not* required to work, and may raise an *AttributeError*.

The Python DOM API, however, *does* require that normal attribute access work. This means that the typical surrogates generated by Python IDL compilers are not likely to work, and wrapper objects may be needed on the client if the DOM objects are accessed via CORBA. While this does require some additional consideration for CORBA DOM clients, the

implementers with experience using DOM over CORBA from Python do not consider this a problem. Attributes that are declared `readonly` may not restrict write access in all DOM implementations.

In the Python DOM API, accessor functions are not required. If provided, they should take the form defined by the Python IDL mapping, but these methods are considered unnecessary since the attributes are accessible directly from Python. "Set" accessors should never be provided for `readonly` attributes.

The IDL definitions do not fully embody the requirements of the W3C DOM API, such as the notion of certain objects, such as the return value of `getElementsByTagName()`, being "live". The Python DOM API does not require implementations to enforce such requirements.

21.7 `xml.dom.minidom` — Minimal DOM implementation

Código Fonte: Lib/xml/dom/minidom.py

`xml.dom.minidom` is a minimal implementation of the Document Object Model interface, with an API similar to that in other languages. It is intended to be simpler than the full DOM and also significantly smaller. Users who are not already proficient with the DOM should consider using the *`xml.etree.ElementTree`* module for their XML processing instead.

> **Aviso:** The *`xml.dom.minidom`* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML vulnerabilities*.

DOM applications typically start by parsing some XML into a DOM. With *`xml.dom.minidom`*, this is done through the parse functions:

```
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml')  # parse an XML file by name

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource)  # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')
```

The *`parse()`* function can take either a filename or an open file object.

`xml.dom.minidom.`**`parse`**(*filename_or_file*, *parser=None*, *bufsize=None*)
Return a `Document` from the given input. *filename_or_file* may be either a file name, or a file-like object. *parser*, if given, must be a SAX2 parser object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the *`parseString()`* function instead:

`xml.dom.minidom.`**`parseString`**(*string*, *parser=None*)
Return a `Document` that represents the *string*. This method creates an *`io.StringIO`* object for the string and passes that on to *`parse()`*.

Both functions return a `Document` object representing the content of the document.

What the *`parse()`* and *`parseString()`* functions do is connect an XML parser with a "DOM builder" that can accept parse events from any SAX parser and convert them into a DOM tree. The name of the functions are perhaps misleading, but are easy to grasp when learning the interfaces. The parsing of the document will be completed before these functions return; it's simply that these functions do not provide a parser implementation themselves.

You can also create a Document by calling a method on a "DOM Implementation" object. You can get this object either by calling the getDOMImplementation() function in the *xml.dom* package or the *xml.dom.minidom* module. Once you have a Document, you can add child nodes to it to populate the DOM:

```
from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification. The main property of the document object is the documentElement property. It gives you the main element in the XML document: the one that holds all others. Here is an example program:

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

When you are finished with a DOM tree, you may optionally call the unlink() method to encourage early cleanup of the now-unneeded objects. unlink() is an *xml.dom.minidom*-specific extension to the DOM API that renders the node and its descendants are essentially useless. Otherwise, Python's garbage collector will eventually take care of the objects in the tree.

Ver também:

Document Object Model (DOM) Level 1 Specification The W3C recommendation for the DOM supported by *xml.dom.minidom*.

21.7.1 Objetos DOM

The definition of the DOM API for Python is given as part of the *xml.dom* module documentation. This section lists the differences between the API and *xml.dom.minidom*.

Node.**unlink**()
: Break internal references within the DOM so that it will be garbage collected on versions of Python without cyclic GC. Even when cyclic GC is available, using this can make large amounts of memory available sooner, so calling this on DOM objects as soon as they are no longer needed is good practice. This only needs to be called on the Document object, but may be called on child nodes to discard children of that node.

  You can avoid calling this method explicitly by using the with statement. The following code will automatically unlink *dom* when the with block is exited:

  ```
  with xml.dom.minidom.parse(datasource) as dom:
      ... # Work with dom.
  ```

Node.**writexml**(*writer*, *indent=""*, *addindent=""*, *newl=""*)
: Write XML to the writer object. The writer receives texts but not bytes as input, it should have a write() method which matches that of the file object interface. The *indent* parameter is the indentation of the current node. The *addindent* parameter is the incremental indentation to use for subnodes of the current one. The *newl* parameter specifies the string to use to terminate newlines.

  For the Document node, an additional keyword argument *encoding* can be used to specify the encoding field of the XML header.

Node.**toxml**(*encoding=None*)
Return a string or byte string containing the XML represented by the DOM node.

With an explicit *encoding*[1] argument, the result is a byte string in the specified encoding. With no *encoding* argument, the result is a Unicode string, and the XML declaration in the resulting string does not specify an encoding. Encoding this string in an encoding other than UTF-8 is likely incorrect, since UTF-8 is the default encoding of XML.

Node.**toprettyxml**(*indent="\t", newl="\n", encoding=None*)
Return a pretty-printed version of the document. *indent* specifies the indentation string and defaults to a tabulator; *newl* specifies the string emitted at the end of each line and defaults to `\n`.

The *encoding* argument behaves like the corresponding argument of *toxml()*.

21.7.2 DOM Example

This example program is a fairly realistic example of a simple program. In this particular case, we do not take much advantage of the flexibility of the DOM.

```
import xml.dom.minidom

document = """\
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print("<html>")
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print("</html>")
```

(continua na próxima página)

[1] The encoding name included in the XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not valid in an XML document's declaration, even though Python accepts it as an encoding name. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

(continuação da página anterior)

```
def handleSlides(slides):
    for slide in slides:
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print("<title>%s</title>" % getText(title.childNodes))

def handleSlideTitle(title):
    print("<h2>%s</h2>" % getText(title.childNodes))

def handlePoints(points):
    print("<ul>")
    for point in points:
        handlePoint(point)
    print("</ul>")

def handlePoint(point):
    print("<li>%s</li>" % getText(point.childNodes))

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print("<p>%s</p>" % getText(title.childNodes))

handleSlideshow(dom)
```

21.7.3 minidom e o padrão DOM

The *xml.dom.minidom* module is essentially a DOM 1.0-compatible DOM with some DOM 2 features (primarily namespace features).

Usage of the DOM interface in Python is straight-forward. The following mapping rules apply:

- Interfaces are accessed through instance objects. Applications should not instantiate the classes themselves; they should use the creator functions available on the `Document` object. Derived interfaces support all operations (and attributes) from the base interfaces, plus any new operations.
- Operations are used as methods. Since the DOM uses only `in` parameters, the arguments are passed in normal order (from left to right). There are no optional arguments. `void` operations return `None`.
- IDL attributes map to instance attributes. For compatibility with the OMG IDL language mapping for Python, an attribute `foo` can also be accessed through accessor methods `_get_foo()` and `_set_foo()`. `readonly` attributes must not be changed; this is not enforced at runtime.
- The types `short int`, `unsigned int`, `unsigned long long`, and `boolean` all map to Python integer objects.
- The type `DOMString` maps to Python strings. *xml.dom.minidom* supports either bytes or strings, but will normally produce strings. Values of type `DOMString` may also be `None` where allowed to have the IDL `null` value by the DOM specification from the W3C.
- `const` declarations map to variables in their respective scope (e.g. `xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE`); they must not be changed.

- `DOMException` is currently not supported in *`xml.dom.minidom`*. Instead, *`xml.dom.minidom`* uses standard Python exceptions such as *`TypeError`* and *`AttributeError`*.
- `NodeList` objects are implemented using Python's built-in list type. These objects provide the interface defined in the DOM specification, but with earlier versions of Python they do not support the official API. They are, however, much more "Pythonic" than the interface defined in the W3C recommendations.

The following interfaces have no implementation in *`xml.dom.minidom`*:

- `DOMTimeStamp`
- `EntityReference`

Most of these reflect information in the XML document that is not of general utility to most DOM users.

21.8 `xml.dom.pulldom` — Support for building partial DOM trees

Código Fonte: Lib/xml/dom/pulldom.py

The *`xml.dom.pulldom`* module provides a "pull parser" which can also be asked to produce DOM-accessible fragments of the document where necessary. The basic concept involves pulling "events" from a stream of incoming XML and processing them. In contrast to SAX which also employs an event-driven processing model together with callbacks, the user of a pull parser is responsible for explicitly pulling events from the stream, looping over those events until either processing is finished or an error condition occurs.

> **Aviso:** The *`xml.dom.pulldom`* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML vulnerabilities*.

Alterado na versão 3.7.1: The SAX parser no longer processes general external entities by default to increase security by default. To enable processing of external entities, pass a custom parser instance in:

```
from xml.dom.pulldom import parse
from xml.sax import make_parser
from xml.sax.handler import feature_external_ges

parser = make_parser()
parser.setFeature(feature_external_ges, True)
parse(filename, parser=parser)
```

Exemplo:

```
from xml.dom import pulldom

doc = pulldom.parse('sales_items.xml')
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'item':
        if int(node.getAttribute('price')) > 50:
            doc.expandNode(node)
            print(node.toxml())
```

`event` is a constant and can be one of:

- `START_ELEMENT`
- `END_ELEMENT`

- COMMENT
- START_DOCUMENT
- END_DOCUMENT
- CHARACTERS
- PROCESSING_INSTRUCTION
- IGNORABLE_WHITESPACE

node is an object of type xml.dom.minidom.Document, xml.dom.minidom.Element or xml.dom.minidom.Text.

Since the document is treated as a "flat" stream of events, the document "tree" is implicitly traversed and the desired elements are found regardless of their depth in the tree. In other words, one does not need to consider hierarchical issues such as recursive searching of the document nodes, although if the context of elements were important, one would either need to maintain some context-related state (i.e. remembering where one is in the document at any given point) or to make use of the *DOMEventStream.expandNode()* method and switch to DOM-related processing.

class xml.dom.pulldom.**PullDom**(*documentFactory=None*)
Subclasse de *xml.sax.handler.ContentHandler*.

class xml.dom.pulldom.**SAX2DOM**(*documentFactory=None*)
Subclasse de *xml.sax.handler.ContentHandler*.

xml.dom.pulldom.**parse**(*stream_or_string*, *parser=None*, *bufsize=None*)
Return a *DOMEventStream* from the given input. *stream_or_string* may be either a file name, or a file-like object. *parser*, if given, must be an *XMLReader* object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the *parseString()* function instead:

xml.dom.pulldom.**parseString**(*string*, *parser=None*)
Return a *DOMEventStream* that represents the (Unicode) *string*.

xml.dom.pulldom.**default_bufsize**
Default value for the *bufsize* parameter to *parse()*.

The value of this variable can be changed before calling *parse()* and the new value will take effect.

21.8.1 Objetos DOMEventStream

class xml.dom.pulldom.**DOMEventStream**(*stream*, *parser*, *bufsize*)

getEvent()
Return a tuple containing *event* and the current *node* as xml.dom.minidom.Document if event equals START_DOCUMENT, xml.dom.minidom.Element if event equals START_ELEMENT or END_ELEMENT or xml.dom.minidom.Text if event equals CHARACTERS. The current node does not contain information about its children, unless *expandNode()* is called.

expandNode(*node*)
Expands all children of *node* into *node*. Example:

```
from xml.dom import pulldom

xml = '<html><title>Foo</title> <p>Some text <div>and more</div></p> </html>'
doc = pulldom.parseString(xml)
```

(continua na próxima página)

(continuação da página anterior)

```
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'p':
        # Following statement only prints '<p/>'
        print(node.toxml())
        doc.expandNode(node)
        # Following statement prints node with all its children '<p>Some text
↪<div>and more</div></p>'
        print(node.toxml())
```

reset()

21.9 xml.sax — Support for SAX2 parsers

Código Fonte: Lib/xml/sax/__init__.py

The *xml.sax* package provides a number of modules which implement the Simple API for XML (SAX) interface for Python. The package itself provides the SAX exceptions and the convenience functions which will be most used by users of the SAX API.

Aviso: The *xml.sax* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML vulnerabilities*.

Alterado na versão 3.7.1: The SAX parser no longer processes general external entities by default to increase security. Before, the parser created network connections to fetch remote files or loaded local files from the file system for DTD and entities. The feature can be enabled again with method *setFeature()* on the parser object and argument *feature_external_ges*.

As funções de conveniência são:

xml.sax.**make_parser**(*parser_list=[]*)
Create and return a SAX *XMLReader* object. The first parser found will be used. If *parser_list* is provided, it must be a list of strings which name modules that have a function named `create_parser()`. Modules listed in *parser_list* will be used before modules in the default list of parsers.

xml.sax.**parse**(*filename_or_stream*, *handler*, *error_handler=handler.ErrorHandler()*)
Create a SAX parser and use it to parse a document. The document, passed in as *filename_or_stream*, can be a filename or a file object. The *handler* parameter needs to be a SAX *ContentHandler* instance. If *error_handler* is given, it must be a SAX *ErrorHandler* instance; if omitted, *SAXParseException* will be raised on all errors. There is no return value; all work must be done by the *handler* passed in.

xml.sax.**parseString**(*string*, *handler*, *error_handler=handler.ErrorHandler()*)
Similar to *parse()*, but parses from a buffer *string* received as a parameter. *string* must be a *str* instance or a *bytes-like object*.

Alterado na versão 3.5: Added support of *str* instances.

A typical SAX application uses three kinds of objects: readers, handlers and input sources. "Reader" in this context is another term for parser, i.e. some piece of code that reads the bytes or characters from the input source, and produces a sequence of events. The events then get distributed to the handler objects, i.e. the reader invokes a method on the handler. A SAX application must therefore obtain a reader object, create or open the input sources, create the handlers, and connect these objects all together. As the final step of preparation, the reader is called to parse the input. During parsing, methods on the handler objects are called based on structural and syntactic events from the input data.

For these objects, only the interfaces are relevant; they are normally not instantiated by the application itself. Since Python does not have an explicit notion of interface, they are formally introduced as classes, but applications may use implementations which do not inherit from the provided classes. The *InputSource*, *Locator*, Attributes, AttributesNS, and *XMLReader* interfaces are defined in the module *xml.sax.xmlreader*. The handler interfaces are defined in *xml.sax.handler*. For convenience, *InputSource* (which is often instantiated directly) and the handler classes are also available from *xml.sax*. These interfaces are described below.

In addition to these classes, *xml.sax* provides the following exception classes.

exception xml.sax.**SAXException**(*msg*, *exception=None*)
Encapsulate an XML error or warning. This class can contain basic error or warning information from either the XML parser or the application: it can be subclassed to provide additional functionality or to add localization. Note that although the handlers defined in the *ErrorHandler* interface receive instances of this exception, it is not required to actually raise the exception — it is also useful as a container for information.

When instantiated, *msg* should be a human-readable description of the error. The optional *exception* parameter, if given, should be None or an exception that was caught by the parsing code and is being passed along as information.

This is the base class for the other SAX exception classes.

exception xml.sax.**SAXParseException**(*msg*, *exception*, *locator*)
Subclass of *SAXException* raised on parse errors. Instances of this class are passed to the methods of the SAX *ErrorHandler* interface to provide information about the parse error. This class supports the SAX *Locator* interface as well as the *SAXException* interface.

exception xml.sax.**SAXNotRecognizedException**(*msg*, *exception=None*)
Subclass of *SAXException* raised when a SAX *XMLReader* is confronted with an unrecognized feature or property. SAX applications and extensions may use this class for similar purposes.

exception xml.sax.**SAXNotSupportedException**(*msg*, *exception=None*)
Subclass of *SAXException* raised when a SAX *XMLReader* is asked to enable a feature that is not supported, or to set a property to a value that the implementation does not support. SAX applications and extensions may use this class for similar purposes.

Ver também:

SAX: The Simple API for XML This site is the focal point for the definition of the SAX API. It provides a Java implementation and online documentation. Links to implementations and historical information are also available.

Module *xml.sax.handler* Definitions of the interfaces for application-provided objects.

Module *xml.sax.saxutils* Convenience functions for use in SAX applications.

Module *xml.sax.xmlreader* Definitions of the interfaces for parser-provided objects.

21.9.1 SAXException Objects

The *SAXException* exception class supports the following methods:

SAXException.**getMessage**()
Return a human-readable message describing the error condition.

SAXException.**getException**()
Return an encapsulated exception object, or None.

21.10 xml.sax.handler — Classes base para manipuladores de SAX

Código Fonte: Lib/xml/sax/handler.py

A API SAX define

class xml.sax.handler.**ContentHandler**
This is the main callback interface in SAX, and the one most important to applications. The order of events in this interface mirrors the order of the information in the document.

class xml.sax.handler.**DTDHandler**
Manipular eventos DTD.

This interface specifies only those DTD events required for basic parsing (unparsed entities and attributes).

class xml.sax.handler.**EntityResolver**
Basic interface for resolving entities. If you create an object implementing this interface, then register the object with your Parser, the parser will call the method in your object to resolve all external entities.

class xml.sax.handler.**ErrorHandler**
Interface used by the parser to present error and warning messages to the application. The methods of this object control whether errors are immediately converted to exceptions or are handled in some other way.

In addition to these classes, *xml.sax.handler* provides symbolic constants for the feature and property names.

xml.sax.handler.**feature_namespaces**

value: "http://xml.org/sax/features/namespaces"
true: Executa o processamento do espaço de nomes.
false: Optionally do not perform Namespace processing (implies namespace-prefixes; default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_namespace_prefixes**

value: "http://xml.org/sax/features/namespace-prefixes"
true: Report the original prefixed names and attributes used for Namespace declarations.
false: Do not report attributes used for Namespace declarations, and optionally do not report original prefixed names (default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_string_interning**

value: "http://xml.org/sax/features/string-interning"
true: All element names, prefixes, attribute names, Namespace URIs, and local names are interned using the built-in intern function.
false: Names are not necessarily interned, although they may be (default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_validation**

value: "http://xml.org/sax/features/validation"
true: Report all validation errors (implies external-general-entities and external-parameter-entities).

false: Do not report validation errors.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_external_ges**

value: "http://xml.org/sax/features/external-general-entities"
true: Include all external general (text) entities.
false: Do not include external general entities.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**feature_external_pes**

value: "http://xml.org/sax/features/external-parameter-entities"
true: Include all external parameter entities, including the external DTD subset.
false: Do not include any external parameter entities, even the external DTD subset.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**all_features**
List of all features.

xml.sax.handler.**property_lexical_handler**

value: "http://xml.org/sax/properties/lexical-handler"
data type: xml.sax.sax2lib.LexicalHandler (not supported in Python 2)
descrição: Um manipulador de extensão opcional para eventos lexicais como comentários.
access: read/write

xml.sax.handler.**property_declaration_handler**

value: "http://xml.org/sax/properties/declaration-handler"
data type: xml.sax.sax2lib.DeclHandler (not supported in Python 2)
description: An optional extension handler for DTD-related events other than notations and unparsed entities.
access: read/write

xml.sax.handler.**property_dom_node**

value: "http://xml.org/sax/properties/dom-node"
data type: org.w3c.dom.Node (not supported in Python 2)
description: When parsing, the current DOM node being visited if this is a DOM iterator; when not parsing, the root DOM node for iteration.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.**property_xml_string**

value: "http://xml.org/sax/properties/xml-string"
tipo de dado: String
description: The literal string of characters that was the source for the current event.
access: read-only

xml.sax.handler.**all_properties**
List of all known property names.

21.10.1 ContentHandler Objects

Users are expected to subclass *ContentHandler* to support their application. The following methods are called by the parser on the appropriate events in the input document:

ContentHandler.**setDocumentLocator**(*locator*)
Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application's business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

ContentHandler.**startDocument**()
Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in DTDHandler (except for *setDocumentLocator()*).

ContentHandler.**endDocument**()
Receive notification of the end of a document.

The SAX parser will invoke this method only once, and it will be the last method invoked during the parse. The parser shall not invoke this method until it has either abandoned parsing (because of an unrecoverable error) or reached the end of input.

ContentHandler.**startPrefixMapping**(*prefix*, *uri*)
Begin the scope of a prefix-URI Namespace mapping.

The information from this event is not necessary for normal Namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the `feature_namespaces` feature is enabled (the default).

There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the *startPrefixMapping()* and *endPrefixMapping()* events supply the information to the application to expand prefixes in those contexts itself, if necessary.

Note that *startPrefixMapping()* and *endPrefixMapping()* events are not guaranteed to be properly nested relative to each-other: all *startPrefixMapping()* events will occur before the corresponding *startElement()* event, and all *endPrefixMapping()* events will occur after the corresponding *endElement()* event, but their order is not guaranteed.

ContentHandler.**endPrefixMapping**(*prefix*)
End the scope of a prefix-URI mapping.

See *startPrefixMapping()* for details. This event will always occur after the corresponding *endElement()* event, but the order of *endPrefixMapping()* events is not otherwise guaranteed.

ContentHandler.**startElement**(*name*, *attrs*)
Signals the start of an element in non-namespace mode.

The *name* parameter contains the raw XML 1.0 name of the element type as a string and the *attrs* parameter holds an object of the `Attributes` interface (see *The Attributes Interface*) containing the attributes of the element. The object passed as *attrs* may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the *copy()* method of the *attrs* object.

ContentHandler.**endElement**(*name*)
Signals the end of an element in non-namespace mode.

The *name* parameter contains the name of the element type, just as with the `startElement()` event.

ContentHandler.**startElementNS**(*name*, *qname*, *attrs*)
Signals the start of an element in namespace mode.

The *name* parameter contains the name of the element type as a `(uri, localname)` tuple, the *qname* parameter contains the raw XML 1.0 name used in the source document, and the *attrs* parameter holds an instance of the `AttributesNS` interface (see *The AttributesNS Interface*) containing the attributes of the element. If no namespace is associated with the element, the *uri* component of *name* will be `None`. The object passed as *attrs* may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the `copy()` method of the *attrs* object.

Parsers may set the *qname* parameter to `None`, unless the `feature_namespace_prefixes` feature is activated.

ContentHandler.**endElementNS**(*name*, *qname*)
Signals the end of an element in namespace mode.

The *name* parameter contains the name of the element type, just as with the `startElementNS()` method, likewise the *qname* parameter.

ContentHandler.**characters**(*content*)
Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information.

content may be a string or bytes instance; the `expat` reader module always produces strings.

Nota: The earlier SAX 1 interface provided by the Python XML Special Interest Group used a more Java-like interface for this method. Since most parsers used from Python did not take advantage of the older interface, the simpler signature was chosen to replace it. To convert old code to the new interface, use *content* instead of slicing content with the old *offset* and *length* parameters.

ContentHandler.**ignorableWhitespace**(*whitespace*)
Receive notification of ignorable whitespace in element content.

Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10): non-validating parsers may also use this method if they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

ContentHandler.**processingInstruction**(*target*, *data*)
Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

ContentHandler.**skippedEntity**(*name*)
Receive notification of a skipped entity.

The Parser will invoke this method once for each entity skipped. Non-validating processors may skip entities if they have not seen the declarations (because, for example, the entity was declared in an external DTD subset). All processors may skip external entities, depending on the values of the `feature_external_ges` and the `feature_external_pes` properties.

21.10.2 DTDHandler Objects

DTDHandler instances provide the following methods:

`DTDHandler.`**`notationDecl`**(*name*, *publicId*, *systemId*)
Handle a notation declaration event.

`DTDHandler.`**`unparsedEntityDecl`**(*name*, *publicId*, *systemId*, *ndata*)
Handle an unparsed entity declaration event.

21.10.3 EntityResolver Objects

`EntityResolver.`**`resolveEntity`**(*publicId*, *systemId*)
Resolve the system identifier of an entity and return either the system identifier to read from as a string, or an InputSource to read from. The default implementation returns *systemId*.

21.10.4 ErrorHandler Objects

Objects with this interface are used to receive error and warning information from the *XMLReader*. If you create an object that implements this interface, then register the object with your *XMLReader*, the parser will call the methods in your object to report all warnings and errors. There are three levels of errors available: warnings, (possibly) recoverable errors, and unrecoverable errors. All methods take a `SAXParseException` as the only parameter. Errors and warnings may be converted to an exception by raising the passed-in exception object.

`ErrorHandler.`**`error`**(*exception*)
Called when the parser encounters a recoverable error. If this method does not raise an exception, parsing may continue, but further document information should not be expected by the application. Allowing the parser to continue may allow additional errors to be discovered in the input document.

`ErrorHandler.`**`fatalError`**(*exception*)
Called when the parser encounters an error it cannot recover from; parsing is expected to terminate when this method returns.

`ErrorHandler.`**`warning`**(*exception*)
Called when the parser presents minor warning information to the application. Parsing is expected to continue when this method returns, and document information will continue to be passed to the application. Raising an exception in this method will cause parsing to end.

21.11 `xml.sax.saxutils` — SAX Utilities

Código Fonte: Lib/xml/sax/saxutils.py

The module *xml.sax.saxutils* contains a number of classes and functions that are commonly useful when creating SAX applications, either in direct use, or as base classes.

`xml.sax.saxutils.`**`escape`**(*data*, *entities={}*)
Escape `'&'`, `'<'`, and `'>'` in a string of data.

You can escape other strings of data by passing a dictionary as the optional *entities* parameter. The keys and values must all be strings; each key will be replaced with its corresponding value. The characters `'&'`, `'<'` and `'>'` are always escaped, even if *entities* is provided.

`xml.sax.saxutils.`**`unescape`**(*data*, *entities={}*)
Unescape `'&'`, `'<'`, and `'>'` in a string of data.

You can unescape other strings of data by passing a dictionary as the optional *entities* parameter. The keys and values must all be strings; each key will be replaced with its corresponding value. `'&amp'`, `'<'`, and `'>'` are always unescaped, even if *entities* is provided.

`xml.sax.saxutils.`**`quoteattr`**(*data*, *entities={}*)
Similar to *escape()*, but also prepares *data* to be used as an attribute value. The return value is a quoted version of *data* with any additional required replacements. *quoteattr()* will select a quote character based on the content of *data*, attempting to avoid encoding any quote characters in the string. If both single- and double-quote characters are already in *data*, the double-quote characters will be encoded and *data* will be wrapped in double-quotes. The resulting string can be used directly as an attribute value:

```
>>> print("<element attr=%s>" % quoteattr("ab ' cd \" ef"))
<element attr="ab ' cd &quot; ef">
```

This function is useful when generating attribute values for HTML or any SGML using the reference concrete syntax.

`class` `xml.sax.saxutils.`**`XMLGenerator`**(*out=None*, *encoding='iso-8859-1'*, *short_empty_elements=False*)
This class implements the *ContentHandler* interface by writing SAX events back into an XML document. In other words, using an *XMLGenerator* as the content handler will reproduce the original document being parsed. *out* should be a file-like object which will default to *sys.stdout*. *encoding* is the encoding of the output stream which defaults to `'iso-8859-1'`. *short_empty_elements* controls the formatting of elements that contain no content: if `False` (the default) they are emitted as a pair of start/end tags, if set to `True` they are emitted as a single self-closed tag.

Novo na versão 3.2: The *short_empty_elements* parameter.

`class` `xml.sax.saxutils.`**`XMLFilterBase`**(*base*)
This class is designed to sit between an *XMLReader* and the client application's event handlers. By default, it does nothing but pass requests up to the reader and events on to the handlers unmodified, but subclasses can override specific methods to modify the event stream or the configuration requests as they pass through.

`xml.sax.saxutils.`**`prepare_input_source`**(*source*, *base=''*)
This function takes an input source and an optional base URL and returns a fully resolved *InputSource* object ready for reading. The input source can be given as a string, a file-like object, or an *InputSource* object; parsers will use this function to implement the polymorphic *source* argument to their `parse()` method.

21.12 xml.sax.xmlreader — Interface for XML parsers

Código Fonte: Lib/xml/sax/xmlreader.py

SAX parsers implement the *XMLReader* interface. They are implemented in a Python module, which must provide a function `create_parser()`. This function is invoked by *xml.sax.make_parser()* with no arguments to create a new parser object.

class xml.sax.xmlreader.**XMLReader**
Base class which can be inherited by SAX parsers.

class xml.sax.xmlreader.**IncrementalParser**
In some cases, it is desirable not to parse an input source at once, but to feed chunks of the document as they get available. Note that the reader will normally not read the entire file, but read it in chunks as well; still `parse()` won't return until the entire document is processed. So these interfaces should be used if the blocking behaviour of `parse()` is not desirable.

When the parser is instantiated it is ready to begin accepting data from the feed method immediately. After parsing has been finished with a call to close the reset method must be called to make the parser ready to accept new data, either from feed or using the parse method.

Note that these methods must *not* be called during parsing, that is, after parse has been called and before it returns.

By default, the class also implements the parse method of the XMLReader interface using the feed, close and reset methods of the IncrementalParser interface as a convenience to SAX 2.0 driver writers.

class xml.sax.xmlreader.**Locator**
Interface for associating a SAX event with a document location. A locator object will return valid results only during calls to DocumentHandler methods; at any other time, the results are unpredictable. If information is not available, methods may return `None`.

class xml.sax.xmlreader.**InputSource**(*system_id=None*)
Encapsulation of the information needed by the *XMLReader* to read entities.

This class may include information about the public identifier, system identifier, byte stream (possibly with character encoding information) and/or the character stream of an entity.

Applications will create objects of this class for use in the *XMLReader.parse()* method and for returning from EntityResolver.resolveEntity.

An *InputSource* belongs to the application, the *XMLReader* is not allowed to modify *InputSource* objects passed to it from the application, although it may make copies and modify those.

class xml.sax.xmlreader.**AttributesImpl**(*attrs*)
This is an implementation of the `Attributes` interface (see section *The Attributes Interface*). This is a dictionary-like object which represents the element attributes in a `startElement()` call. In addition to the most useful dictionary operations, it supports a number of other methods as described by the interface. Objects of this class should be instantiated by readers; *attrs* must be a dictionary-like object containing a mapping from attribute names to attribute values.

class xml.sax.xmlreader.**AttributesNSImpl**(*attrs*, *qnames*)
Namespace-aware variant of *AttributesImpl*, which will be passed to `startElementNS()`. It is derived from *AttributesImpl*, but understands attribute names as two-tuples of *namespaceURI* and *localname*. In addition, it provides a number of methods expecting qualified names as they appear in the original document. This class implements the `AttributesNS` interface (see section *The AttributesNS Interface*).

21.12.1 XMLReader Objects

The *XMLReader* interface supports the following methods:

XMLReader.**parse**(*source*)
Process an input source, producing SAX events. The *source* object can be a system identifier (a string identifying the input source – typically a file name or a URL), a file-like object, or an *InputSource* object. When *parse()* returns, the input is completely processed, and the parser object can be discarded or reset.

Alterado na versão 3.5: Added support of character streams.

XMLReader.**getContentHandler**()
Return the current *ContentHandler*.

XMLReader.**setContentHandler**(*handler*)
Set the current *ContentHandler*. If no *ContentHandler* is set, content events will be discarded.

XMLReader.**getDTDHandler**()
Return the current *DTDHandler*.

XMLReader.**setDTDHandler**(*handler*)
Set the current *DTDHandler*. If no *DTDHandler* is set, DTD events will be discarded.

XMLReader.**getEntityResolver**()
Return the current *EntityResolver*.

XMLReader.**setEntityResolver**(*handler*)
Set the current *EntityResolver*. If no *EntityResolver* is set, attempts to resolve an external entity will result in opening the system identifier for the entity, and fail if it is not available.

XMLReader.**getErrorHandler**()
Return the current *ErrorHandler*.

XMLReader.**setErrorHandler**(*handler*)
Set the current error handler. If no *ErrorHandler* is set, errors will be raised as exceptions, and warnings will be printed.

XMLReader.**setLocale**(*locale*)
Allow an application to set the locale for errors and warnings.

SAX parsers are not required to provide localization for errors and warnings; if they cannot support the requested locale, however, they must raise a SAX exception. Applications may request a locale change in the middle of a parse.

XMLReader.**getFeature**(*featurename*)
Return the current setting for feature *featurename*. If the feature is not recognized, `SAXNotRecognizedException` is raised. The well-known featurenames are listed in the module *xml.sax.handler*.

XMLReader.**setFeature**(*featurename*, *value*)
Set the *featurename* to *value*. If the feature is not recognized, `SAXNotRecognizedException` is raised. If the feature or its setting is not supported by the parser, *SAXNotSupportedException* is raised.

XMLReader.**getProperty**(*propertyname*)
Return the current setting for property *propertyname*. If the property is not recognized, a `SAXNotRecognizedException` is raised. The well-known propertynames are listed in the module *xml.sax.handler*.

XMLReader.**setProperty**(*propertyname*, *value*)
Set the *propertyname* to *value*. If the property is not recognized, `SAXNotRecognizedException` is raised. If the property or its setting is not supported by the parser, *SAXNotSupportedException* is raised.

21.12.2 IncrementalParser Objects

Instances of *IncrementalParser* offer the following additional methods:

IncrementalParser.**feed**(*data*)
Process a chunk of *data*.

IncrementalParser.**close**()
Assume the end of the document. That will check well-formedness conditions that can be checked only at the end, invoke handlers, and may clean up resources allocated during parsing.

IncrementalParser.**reset**()
This method is called after close has been called to reset the parser so that it is ready to parse new documents. The results of calling parse or feed after close without calling reset are undefined.

21.12.3 Locator Objects

Instances of *Locator* provide these methods:

Locator.**getColumnNumber**()
Return the column number where the current event begins.

Locator.**getLineNumber**()
Return the line number where the current event begins.

Locator.**getPublicId**()
Return the public identifier for the current event.

Locator.**getSystemId**()
Return the system identifier for the current event.

21.12.4 InputSource Objects

InputSource.**setPublicId**(*id*)
Sets the public identifier of this *InputSource*.

InputSource.**getPublicId**()
Returns the public identifier of this *InputSource*.

InputSource.**setSystemId**(*id*)
Sets the system identifier of this *InputSource*.

InputSource.**getSystemId**()
Returns the system identifier of this *InputSource*.

InputSource.**setEncoding**(*encoding*)
Sets the character encoding of this *InputSource*.

The encoding must be a string acceptable for an XML encoding declaration (see section 4.3.3 of the XML recommendation).

The encoding attribute of the *InputSource* is ignored if the *InputSource* also contains a character stream.

InputSource.**getEncoding**()
Obtém a codificação de caracteres deste InputSource.

InputSource.**setByteStream**(*bytefile*)
Set the byte stream (a *binary file*) for this input source.

The SAX parser will ignore this if there is also a character stream specified, but it will use a byte stream in preference to opening a URI connection itself.

If the application knows the character encoding of the byte stream, it should set it with the setEncoding method.

`InputSource.`**`getByteStream`**`()`
Get the byte stream for this input source.

The getEncoding method will return the character encoding for this byte stream, or `None` if unknown.

`InputSource.`**`setCharacterStream`**`(`*charfile*`)`
Set the character stream (a *text file*) for this input source.

If there is a character stream specified, the SAX parser will ignore any byte stream and will not attempt to open a URI connection to the system identifier.

`InputSource.`**`getCharacterStream`**`()`
Get the character stream for this input source.

21.12.5 The `Attributes` Interface

`Attributes` objects implement a portion of the *mapping protocol*, including the methods `copy()`, `get()`, `__contains__()`, `items()`, `keys()`, and `values()`. The following methods are also provided:

`Attributes.`**`getLength`**`()`
Return the number of attributes.

`Attributes.`**`getNames`**`()`
Return the names of the attributes.

`Attributes.`**`getType`**`(`*name*`)`
Returns the type of the attribute *name*, which is normally `'CDATA'`.

`Attributes.`**`getValue`**`(`*name*`)`
Return the value of attribute *name*.

21.12.6 The `AttributesNS` Interface

This interface is a subtype of the `Attributes` interface (see section *The Attributes Interface*). All methods supported by that interface are also available on `AttributesNS` objects.

The following methods are also available:

`AttributesNS.`**`getValueByQName`**`(`*name*`)`
Return the value for a qualified name.

`AttributesNS.`**`getNameByQName`**`(`*name*`)`
Return the `(namespace, localname)` pair for a qualified *name*.

`AttributesNS.`**`getQNameByName`**`(`*name*`)`
Return the qualified name for a `(namespace, localname)` pair.

`AttributesNS.`**`getQNames`**`()`
Return the qualified names of all attributes.

21.13 xml.parsers.expat — Fast XML parsing using Expat

Aviso: The pyexpat module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML vulnerabilities*.

The *xml.parsers.expat* module is a Python interface to the Expat non-validating XML parser. The module provides a single extension type, xmlparser, that represents the current state of an XML parser. After an xmlparser object has been created, various attributes of the object can be set to handler functions. When an XML document is then fed to the parser, the handler functions are called for the character data and markup in the XML document.

This module uses the pyexpat module to provide access to the Expat parser. Direct use of the pyexpat module is deprecated.

This module provides one exception and one type object:

exception xml.parsers.expat.**ExpatError**
The exception raised when Expat reports an error. See section *Exceções ExpatError* for more information on interpreting Expat errors.

exception xml.parsers.expat.**error**
Alias for *ExpatError*.

xml.parsers.expat.**XMLParserType**
The type of the return values from the *ParserCreate()* function.

The *xml.parsers.expat* module contains two functions:

xml.parsers.expat.**ErrorString**(*errno*)
Returns an explanatory string for a given error number *errno*.

xml.parsers.expat.**ParserCreate**(*encoding=None*, *namespace_separator=None*)
Creates and returns a new xmlparser object. *encoding*, if specified, must be a string naming the encoding used by the XML data. Expat doesn't support as many encodings as Python does, and its repertoire of encodings can't be extended; it supports UTF-8, UTF-16, ISO-8859-1 (Latin1), and ASCII. If *encoding*[1] is given it will override the implicit or explicit encoding of the document.

Expat can optionally do XML namespace processing for you, enabled by providing a value for *namespace_separator*. The value must be a one-character string; a *ValueError* will be raised if the string has an illegal length (None is considered the same as omission). When namespace processing is enabled, element type names and attribute names that belong to a namespace will be expanded. The element name passed to the element handlers StartElementHandler and EndElementHandler will be the concatenation of the namespace URI, the namespace separator character, and the local part of the name. If the namespace separator is a zero byte (chr(0)) then the namespace URI and the local part will be concatenated without any separator.

For example, if *namespace_separator* is set to a space character (' ') and the following document is parsed:

```
<?xml version="1.0"?>
<root xmlns    = "http://default-namespace.org/"
      xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

[1] The encoding string included in XML output should conform to the appropriate standards. For example, "UTF-8" is valid, but "UTF8" is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

`StartElementHandler` will receive the following strings for each element:

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

Due to limitations in the `Expat` library used by `pyexpat`, the `xmlparser` instance returned can only be used to parse a single XML document. Call `ParserCreate` for each document to provide unique parser instances.

Ver também:

The Expat XML Parser Home page of the Expat project.

21.13.1 Objetos XMLParser

`xmlparser` objects have the following methods:

`xmlparser.`**`Parse`**(*data*[, *isfinal*])
Parses the contents of the string *data*, calling the appropriate handler functions to process the parsed data. *isfinal* must be true on the final call to this method; it allows the parsing of a single file in fragments, not the submission of multiple files. *data* can be the empty string at any time.

`xmlparser.`**`ParseFile`**(*file*)
Parse XML data reading from the object *file*. *file* only needs to provide the `read(nbytes)` method, returning the empty string when there's no more data.

`xmlparser.`**`SetBase`**(*base*)
Sets the base to be used for resolving relative URIs in system identifiers in declarations. Resolving relative identifiers is left to the application: this value will be passed through as the *base* argument to the *`ExternalEntityRefHandler()`*, *`NotationDeclHandler()`*, and *`UnparsedEntityDeclHandler()`* functions.

`xmlparser.`**`GetBase`**()
Returns a string containing the base set by a previous call to *`SetBase()`*, or `None` if *`SetBase()`* hasn't been called.

`xmlparser.`**`GetInputContext`**()
Returns the input data that generated the current event as a string. The data is in the encoding of the entity which contains the text. When called while an event handler is not active, the return value is `None`.

`xmlparser.`**`ExternalEntityParserCreate`**(*context*[, *encoding*])
Create a "child" parser which can be used to parse an external parsed entity referred to by content parsed by the parent parser. The *context* parameter should be the string passed to the *`ExternalEntityRefHandler()`* handler function, described below. The child parser is created with the *`ordered_attributes`* and *`specified_attributes`* set to the values of this parser.

`xmlparser.`**`SetParamEntityParsing`**(*flag*)
Control parsing of parameter entities (including the external DTD subset). Possible *flag* values are `XML_PARAM_ENTITY_PARSING_NEVER`, `XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE` and `XML_PARAM_ENTITY_PARSING_ALWAYS`. Return true if setting the flag was successful.

`xmlparser.`**`UseForeignDTD`**([*flag*])
Calling this with a true value for *flag* (the default) will cause Expat to call the *`ExternalEntityRefHandler`* with *`None`* for all arguments to allow an alternate DTD to be loaded. If the document does not contain a document type declaration, the *`ExternalEntityRefHandler`* will still be called, but the *`StartDoctypeDeclHandler`* and *`EndDoctypeDeclHandler`* will not be called.

Passing a false value for *flag* will cancel a previous call that passed a true value, but otherwise has no effect.

This method can only be called before the *Parse()* or *ParseFile()* methods are called; calling it after either of those have been called causes *ExpatError* to be raised with the *code* attribute set to `errors.codes[errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING]`.

`xmlparser` objects have the following attributes:

`xmlparser.`**buffer_size**
The size of the buffer used when *buffer_text* is true. A new buffer size can be set by assigning a new integer value to this attribute. When the size is changed, the buffer will be flushed.

`xmlparser.`**buffer_text**
Setting this to true causes the `xmlparser` object to buffer textual content returned by Expat to avoid multiple calls to the *CharacterDataHandler()* callback whenever possible. This can improve performance substantially since Expat normally breaks character data into chunks at every line ending. This attribute is false by default, and may be changed at any time.

`xmlparser.`**buffer_used**
If *buffer_text* is enabled, the number of bytes stored in the buffer. These bytes represent UTF-8 encoded text. This attribute has no meaningful interpretation when *buffer_text* is false.

`xmlparser.`**ordered_attributes**
Setting this attribute to a non-zero integer causes the attributes to be reported as a list rather than a dictionary. The attributes are presented in the order found in the document text. For each attribute, two list entries are presented: the attribute name and the attribute value. (Older versions of this module also used this format.) By default, this attribute is false; it may be changed at any time.

`xmlparser.`**specified_attributes**
If set to a non-zero integer, the parser will report only those attributes which were specified in the document instance and not those which were derived from attribute declarations. Applications which set this need to be especially careful to use what additional information is available from the declarations as needed to comply with the standards for the behavior of XML processors. By default, this attribute is false; it may be changed at any time.

The following attributes contain values relating to the most recent error encountered by an `xmlparser` object, and will only have correct values once a call to `Parse()` or `ParseFile()` has raised an *xml.parsers.expat.ExpatError* exception.

`xmlparser.`**ErrorByteIndex**
Byte index at which an error occurred.

`xmlparser.`**ErrorCode**
Numeric code specifying the problem. This value can be passed to the *ErrorString()* function, or compared to one of the constants defined in the `errors` object.

`xmlparser.`**ErrorColumnNumber**
Column number at which an error occurred.

`xmlparser.`**ErrorLineNumber**
Line number at which an error occurred.

The following attributes contain values relating to the current parse location in an `xmlparser` object. During a callback reporting a parse event they indicate the location of the first of the sequence of characters that generated the event. When called outside of a callback, the position indicated will be just past the last parse event (regardless of whether there was an associated callback).

`xmlparser.`**CurrentByteIndex**
Current byte index in the parser input.

`xmlparser.`**CurrentColumnNumber**
Current column number in the parser input.

xmlparser.**CurrentLineNumber**
Current line number in the parser input.

Here is the list of handlers that can be set. To set a handler on an xmlparser object *o*, use o.handlername = func. *handlername* must be taken from the following list, and *func* must be a callable object accepting the correct number of arguments. The arguments are all strings, unless otherwise stated.

xmlparser.**XmlDeclHandler**(*version*, *encoding*, *standalone*)
Called when the XML declaration is parsed. The XML declaration is the (optional) declaration of the applicable version of the XML recommendation, the encoding of the document text, and an optional "standalone" declaration. *version* and *encoding* will be strings, and *standalone* will be 1 if the document is declared standalone, 0 if it is declared not to be standalone, or -1 if the standalone clause was omitted. This is only available with Expat version 1.95.0 or newer.

xmlparser.**StartDoctypeDeclHandler**(*doctypeName*, *systemId*, *publicId*, *has_internal_subset*)
Called when Expat begins parsing the document type declaration (<!DOCTYPE ...). The *doctypeName* is provided exactly as presented. The *systemId* and *publicId* parameters give the system and public identifiers if specified, or None if omitted. *has_internal_subset* will be true if the document contains and internal document declaration subset. This requires Expat version 1.2 or newer.

xmlparser.**EndDoctypeDeclHandler**()
Called when Expat is done parsing the document type declaration. This requires Expat version 1.2 or newer.

xmlparser.**ElementDeclHandler**(*name*, *model*)
Called once for each element type declaration. *name* is the name of the element type, and *model* is a representation of the content model.

xmlparser.**AttlistDeclHandler**(*elname*, *attname*, *type*, *default*, *required*)
Called for each declared attribute for an element type. If an attribute list declaration declares three attributes, this handler is called three times, once for each attribute. *elname* is the name of the element to which the declaration applies and *attname* is the name of the attribute declared. The attribute type is a string passed as *type*; the possible values are 'CDATA', 'ID', 'IDREF', ... *default* gives the default value for the attribute used when the attribute is not specified by the document instance, or None if there is no default value (#IMPLIED values). If the attribute is required to be given in the document instance, *required* will be true. This requires Expat version 1.95.0 or newer.

xmlparser.**StartElementHandler**(*name*, *attributes*)
Called for the start of every element. *name* is a string containing the element name, and *attributes* is the element attributes. If *ordered_attributes* is true, this is a list (see *ordered_attributes* for a full description). Otherwise it's a dictionary mapping names to values.

xmlparser.**EndElementHandler**(*name*)
Called for the end of every element.

xmlparser.**ProcessingInstructionHandler**(*target*, *data*)
Called for every processing instruction.

xmlparser.**CharacterDataHandler**(*data*)
Called for character data. This will be called for normal character data, CDATA marked content, and ignorable whitespace. Applications which must distinguish these cases can use the *StartCdataSectionHandler*, *EndCdataSectionHandler*, and *ElementDeclHandler* callbacks to collect the required information.

xmlparser.**UnparsedEntityDeclHandler**(*entityName*, *base*, *systemId*, *publicId*, *notationName*)
Called for unparsed (NDATA) entity declarations. This is only present for version 1.2 of the Expat library; for more recent versions, use *EntityDeclHandler* instead. (The underlying function in the Expat library has been declared obsolete.)

xmlparser.**EntityDeclHandler**(*entityName*, *is_parameter_entity*, *value*, *base*, *systemId*, *publicId*, *notationName*)
Called for all entity declarations. For parameter and internal entities, *value* will be a string giving the declared contents of the entity; this will be None for external entities. The *notationName* parameter will be None for

parsed entities, and the name of the notation for unparsed entities. *is_parameter_entity* will be true if the entity is a parameter entity or false for general entities (most applications only need to be concerned with general entities). This is only available starting with version 1.95.0 of the Expat library.

xmlparser.**NotationDeclHandler**(*notationName*, *base*, *systemId*, *publicId*)
Called for notation declarations. *notationName*, *base*, and *systemId*, and *publicId* are strings if given. If the public identifier is omitted, *publicId* will be `None`.

xmlparser.**StartNamespaceDeclHandler**(*prefix*, *uri*)
Called when an element contains a namespace declaration. Namespace declarations are processed before the *StartElementHandler* is called for the element on which declarations are placed.

xmlparser.**EndNamespaceDeclHandler**(*prefix*)
Called when the closing tag is reached for an element that contained a namespace declaration. This is called once for each namespace declaration on the element in the reverse of the order for which the *StartNamespaceDeclHandler* was called to indicate the start of each namespace declaration's scope. Calls to this handler are made after the corresponding *EndElementHandler* for the end of the element.

xmlparser.**CommentHandler**(*data*)
Called for comments. *data* is the text of the comment, excluding the leading `'<!--'` and trailing `'-->'`.

xmlparser.**StartCdataSectionHandler**()
Called at the start of a CDATA section. This and *EndCdataSectionHandler* are needed to be able to identify the syntactical start and end for CDATA sections.

xmlparser.**EndCdataSectionHandler**()
Called at the end of a CDATA section.

xmlparser.**DefaultHandler**(*data*)
Called for any characters in the XML document for which no applicable handler has been specified. This means characters that are part of a construct which could be reported, but for which no handler has been supplied.

xmlparser.**DefaultHandlerExpand**(*data*)
This is the same as the *DefaultHandler()*, but doesn't inhibit expansion of internal entities. The entity reference will not be passed to the default handler.

xmlparser.**NotStandaloneHandler**()
Called if the XML document hasn't been declared as being a standalone document. This happens when there is an external subset or a reference to a parameter entity, but the XML declaration does not set standalone to `yes` in an XML declaration. If this handler returns `0`, then the parser will raise an `XML_ERROR_NOT_STANDALONE` error. If this handler is not set, no exception is raised by the parser for this condition.

xmlparser.**ExternalEntityRefHandler**(*context*, *base*, *systemId*, *publicId*)
Called for references to external entities. *base* is the current base, as set by a previous call to *SetBase()*. The public and system identifiers, *systemId* and *publicId*, are strings if given; if the public identifier is not given, *publicId* will be `None`. The *context* value is opaque and should only be used as described below.

For external entities to be parsed, this handler must be implemented. It is responsible for creating the sub-parser using `ExternalEntityParserCreate(context)`, initializing it with the appropriate callbacks, and parsing the entity. This handler should return an integer; if it returns `0`, the parser will raise an `XML_ERROR_EXTERNAL_ENTITY_HANDLING` error, otherwise parsing will continue.

If this handler is not provided, external entities are reported by the *DefaultHandler* callback, if provided.

21.13.2 Exceções ExpatError

ExpatError exceptions have a number of interesting attributes:

ExpatError.**code**
: Expat's internal error number for the specific error. The *errors.messages* dictionary maps these error numbers to Expat's error messages. For example:

```
from xml.parsers.expat import ParserCreate, ExpatError, errors

p = ParserCreate()
try:
    p.Parse(some_xml_document)
except ExpatError as err:
    print("Error:", errors.messages[err.code])
```

The *errors* module also provides error message constants and a dictionary *codes* mapping these messages back to the error codes, see below.

ExpatError.**lineno**
: Line number on which the error was detected. The first line is numbered `1`.

ExpatError.**offset**
: Character offset into the line where the error occurred. The first column is numbered `0`.

21.13.3 Exemplo

The following program defines three handlers that just print out their arguments.

```
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print('Start element:', name, attrs)
def end_element(name):
    print('End element:', name)
def char_data(data):
    print('Character data:', repr(data))

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

The output from this program is:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
```

(continua na próxima página)

(continuação da página anterior)

```
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

21.13.4 Content Model Descriptions

Content models are described using nested tuples. Each tuple contains four values: the type, the quantifier, the name, and a tuple of children. Children are simply additional content model descriptions.

The values of the first two fields are constants defined in the *xml.parsers.expat.model* module. These constants can be collected in two groups: the model type group and the quantifier group.

The constants in the model type group are:

xml.parsers.expat.model.**XML_CTYPE_ANY**
The element named by the model name was declared to have a content model of `ANY`.

xml.parsers.expat.model.**XML_CTYPE_CHOICE**
The named element allows a choice from a number of options; this is used for content models such as `(A | B | C)`.

xml.parsers.expat.model.**XML_CTYPE_EMPTY**
Elements which are declared to be `EMPTY` have this model type.

xml.parsers.expat.model.**XML_CTYPE_MIXED**

xml.parsers.expat.model.**XML_CTYPE_NAME**

xml.parsers.expat.model.**XML_CTYPE_SEQ**
Models which represent a series of models which follow one after the other are indicated with this model type. This is used for models such as `(A, B, C)`.

The constants in the quantifier group are:

xml.parsers.expat.model.**XML_CQUANT_NONE**
No modifier is given, so it can appear exactly once, as for `A`.

xml.parsers.expat.model.**XML_CQUANT_OPT**
The model is optional: it can appear once or not at all, as for `A?`.

xml.parsers.expat.model.**XML_CQUANT_PLUS**
The model must occur one or more times (like `A+`).

xml.parsers.expat.model.**XML_CQUANT_REP**
The model must occur zero or more times, as for `A*`.

21.13.5 Expat error constants

The following constants are provided in the *xml.parsers.expat.errors* module. These constants are useful in interpreting some of the attributes of the ExpatError exception objects raised when an error has occurred. Since for backwards compatibility reasons, the constants' value is the error *message* and not the numeric error *code*, you do this by comparing its *code* attribute with `errors.codes[errors.XML_ERROR_`*CONSTANT_NAME*`]`.

The `errors` module has the following attributes:

xml.parsers.expat.errors.**codes**
A dictionary mapping numeric error codes to their string descriptions.

Novo na versão 3.2.

xml.parsers.expat.errors.**messages**
A dictionary mapping string descriptions to their error codes.

Novo na versão 3.2.

xml.parsers.expat.errors.**XML_ERROR_ASYNC_ENTITY**

xml.parsers.expat.errors.**XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF**
An entity reference in an attribute value referred to an external entity instead of an internal entity.

xml.parsers.expat.errors.**XML_ERROR_BAD_CHAR_REF**
A character reference referred to a character which is illegal in XML (for example, character `0`, or '`&#0;`').

xml.parsers.expat.errors.**XML_ERROR_BINARY_ENTITY_REF**
An entity reference referred to an entity which was declared with a notation, so cannot be parsed.

xml.parsers.expat.errors.**XML_ERROR_DUPLICATE_ATTRIBUTE**
An attribute was used more than once in a start tag.

xml.parsers.expat.errors.**XML_ERROR_INCORRECT_ENCODING**

xml.parsers.expat.errors.**XML_ERROR_INVALID_TOKEN**
Raised when an input byte could not properly be assigned to a character; for example, a NUL byte (value `0`) in a UTF-8 input stream.

xml.parsers.expat.errors.**XML_ERROR_JUNK_AFTER_DOC_ELEMENT**
Something other than whitespace occurred after the document element.

xml.parsers.expat.errors.**XML_ERROR_MISPLACED_XML_PI**
An XML declaration was found somewhere other than the start of the input data.

xml.parsers.expat.errors.**XML_ERROR_NO_ELEMENTS**
The document contains no elements (XML requires all documents to contain exactly one top-level element)..

xml.parsers.expat.errors.**XML_ERROR_NO_MEMORY**
Expat was not able to allocate memory internally.

xml.parsers.expat.errors.**XML_ERROR_PARAM_ENTITY_REF**
A parameter entity reference was found where it was not allowed.

xml.parsers.expat.errors.**XML_ERROR_PARTIAL_CHAR**
An incomplete character was found in the input.

xml.parsers.expat.errors.**XML_ERROR_RECURSIVE_ENTITY_REF**
An entity reference contained another reference to the same entity; possibly via a different name, and possibly indirectly.

xml.parsers.expat.errors.**XML_ERROR_SYNTAX**
Some unspecified syntax error was encountered.

xml.parsers.expat.errors.**XML_ERROR_TAG_MISMATCH**
An end tag did not match the innermost open start tag.

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_TOKEN**
Some token (such as a start tag) was not closed before the end of the stream or the next token was encountered.

xml.parsers.expat.errors.**XML_ERROR_UNDEFINED_ENTITY**
A reference was made to an entity which was not defined.

xml.parsers.expat.errors.**XML_ERROR_UNKNOWN_ENCODING**
The document encoding is not supported by Expat.

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_CDATA_SECTION**
A CDATA marked section was not closed.

xml.parsers.expat.errors.**XML_ERROR_EXTERNAL_ENTITY_HANDLING**

xml.parsers.expat.errors.**XML_ERROR_NOT_STANDALONE**
The parser determined that the document was not "standalone" though it declared itself to be in the XML declaration, and the `NotStandaloneHandler` was set and returned `0`.

xml.parsers.expat.errors.**XML_ERROR_UNEXPECTED_STATE**

xml.parsers.expat.errors.**XML_ERROR_ENTITY_DECLARED_IN_PE**

xml.parsers.expat.errors.**XML_ERROR_FEATURE_REQUIRES_XML_DTD**
An operation was requested that requires DTD support to be compiled in, but Expat was configured without DTD support. This should never be reported by a standard build of the *xml.parsers.expat* module.

xml.parsers.expat.errors.**XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING**
A behavioral change was requested after parsing started that can only be changed before parsing has started. This is (currently) only raised by `UseForeignDTD()`.

xml.parsers.expat.errors.**XML_ERROR_UNBOUND_PREFIX**
An undeclared prefix was found when namespace processing was enabled.

xml.parsers.expat.errors.**XML_ERROR_UNDECLARING_PREFIX**
The document attempted to remove the namespace declaration associated with a prefix.

xml.parsers.expat.errors.**XML_ERROR_INCOMPLETE_PE**
A parameter entity contained incomplete markup.

xml.parsers.expat.errors.**XML_ERROR_XML_DECL**
The document contained no document element at all.

xml.parsers.expat.errors.**XML_ERROR_TEXT_DECL**
There was an error parsing a text declaration in an external entity.

xml.parsers.expat.errors.**XML_ERROR_PUBLICID**
Characters were found in the public id that are not allowed.

xml.parsers.expat.errors.**XML_ERROR_SUSPENDED**
The requested operation was made on a suspended parser, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.errors.**XML_ERROR_NOT_SUSPENDED**
An attempt to resume the parser was made when the parser had not been suspended.

xml.parsers.expat.errors.**XML_ERROR_ABORTED**
This should not be reported to Python applications.

xml.parsers.expat.errors.**XML_ERROR_FINISHED**
The requested operation was made on a parser which was finished parsing input, but isn't allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.errors.**XML_ERROR_SUSPEND_PE**

CAPÍTULO 22

Protocolos de Internet e Suporte

Os módulos descritos neste capítulo implementam protocolos de Internet e suporte para tecnologias relacionadas. Todos eles estão implementados em Python. A maioria destes módulos requer a presença do módulo dependente do sistema *socket*, que é suportado atualmente na maioria das plataformas populares. Aqui temos uma visão geral:

22.1 webbrowser — Convenient Web-browser controller

Código Fonte: Lib/webbrowser.py

The *webbrowser* module provides a high-level interface to allow displaying Web-based documents to users. Under most circumstances, simply calling the *open()* function from this module will do the right thing.

Under Unix, graphical browsers are preferred under X11, but text-mode browsers will be used if graphical browsers are not available or an X11 display isn't available. If text-mode browsers are used, the calling process will block until the user exits the browser.

If the environment variable BROWSER exists, it is interpreted as the *os.pathsep*-separated list of browsers to try ahead of the platform defaults. When the value of a list part contains the string %s, then it is interpreted as a literal browser command line to be used with the argument URL substituted for %s; if the part does not contain %s, it is simply interpreted as the name of the browser to launch.[1]

For non-Unix platforms, or when a remote browser is available on Unix, the controlling process will not wait for the user to finish with the browser, but allow the remote browser to maintain its own windows on the display. If remote browsers are not available on Unix, the controlling process will launch a new browser and wait.

The script **webbrowser** can be used as a command-line interface for the module. It accepts a URL as the argument. It accepts the following optional parameters: -n opens the URL in a new browser window, if possible; -t opens the URL in a new browser page ("tab"). The options are, naturally, mutually exclusive. Usage example:

```
python -m webbrowser -t "http://www.python.org"
```

[1] Executables named here without a full path will be searched in the directories given in the PATH environment variable.

The following exception is defined:

exception webbrowser.**Error**
Exception raised when a browser control error occurs.

As seguintes funções são definidas:

webbrowser.**open**(*url*, *new=0*, *autoraise=True*)
Display *url* using the default browser. If *new* is 0, the *url* is opened in the same browser window if possible. If *new* is 1, a new browser window is opened if possible. If *new* is 2, a new browser page ("tab") is opened if possible. If *autoraise* is `True`, the window is raised if possible (note that under many window managers this will occur regardless of the setting of this variable).

Note that on some platforms, trying to open a filename using this function, may work and start the operating system's associated program. However, this is neither supported nor portable.

webbrowser.**open_new**(*url*)
Open *url* in a new window of the default browser, if possible, otherwise, open *url* in the only browser window.

webbrowser.**open_new_tab**(*url*)
Open *url* in a new page ("tab") of the default browser, if possible, otherwise equivalent to *open_new()*.

webbrowser.**get**(*using=None*)
Return a controller object for the browser type *using*. If *using* is `None`, return a controller for a default browser appropriate to the caller's environment.

webbrowser.**register**(*name*, *constructor*, *instance=None*, ***, *preferred=False*)
Register the browser type *name*. Once a browser type is registered, the *get()* function can return a controller for that browser type. If *instance* is not provided, or is `None`, *constructor* will be called without parameters to create an instance when needed. If *instance* is provided, *constructor* will never be called, and may be `None`.

Setting *preferred* to `True` makes this browser a preferred result for a *get()* call with no argument. Otherwise, this entry point is only useful if you plan to either set the `BROWSER` variable or call *get()* with a nonempty argument matching the name of a handler you declare.

Alterado na versão 3.7: *preferred* keyword-only parameter was added.

A number of browser types are predefined. This table gives the type names that may be passed to the *get()* function and the corresponding instantiations for the controller classes, all defined in this module.

Type Name	Nome da Classe	Notas
`'mozilla'`	`Mozilla('mozilla')`	
`'firefox'`	`Mozilla('mozilla')`	
`'netscape'`	`Mozilla('netscape')`	
`'galeon'`	`Galeon('galeon')`	
`'epiphany'`	`Galeon('epiphany')`	
`'skipstone'`	`BackgroundBrowser('skipstone')`	
`'kfmclient'`	`Konqueror()`	(1)
`'konqueror'`	`Konqueror()`	(1)
`'kfm'`	`Konqueror()`	(1)
`'mosaic'`	`BackgroundBrowser('mosaic')`	
`'opera'`	`Opera()`	
`'grail'`	`Grail()`	
`'links'`	`GenericBrowser('links')`	
`'elinks'`	`Elinks('elinks')`	
`'lynx'`	`GenericBrowser('lynx')`	
`'w3m'`	`GenericBrowser('w3m')`	
`'windows-default'`	`WindowsDefault`	(2)
`'macosx'`	`MacOSX('default')`	(3)
`'safari'`	`MacOSX('safari')`	(3)
`'google-chrome'`	`Chrome('google-chrome')`	
`'chrome'`	`Chrome('chrome')`	
`'chromium'`	`Chromium('chromium')`	
`'chromium-browser'`	`Chromium('chromium-browser')`	

Notas:

(1) "Konqueror" is the file manager for the KDE desktop environment for Unix, and only makes sense to use if KDE is running. Some way of reliably detecting KDE would be nice; the `KDEDIR` variable is not sufficient. Note also that the name "kfm" is used even when using the **`konqueror`** command with KDE 2 — the implementation selects the best strategy for running Konqueror.

(2) Somente em Plataformas Windows.

(3) Somente na plataforma Mac OS X.

Novo na versão 3.3: Support for Chrome/Chromium has been added.

Aqui estão alguns exemplos simples:

```
url = 'http://docs.python.org/'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url)

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

22.1.1 Browser Controller Objects

Browser controllers provide these methods which parallel three of the module-level convenience functions:

`controller.`**`open`**(*url*, *new=0*, *autoraise=True*)
Display *url* using the browser handled by this controller. If *new* is 1, a new browser window is opened if possible. If *new* is 2, a new browser page ("tab") is opened if possible.

`controller.`**`open_new`**(*url*)
Open *url* in a new window of the browser handled by this controller, if possible, otherwise, open *url* in the only browser window. Alias *open_new()*.

`controller.`**`open_new_tab`**(*url*)
Open *url* in a new page ("tab") of the browser handled by this controller, if possible, otherwise equivalent to *open_new()*.

22.2 `cgi` — Suporte a Common Gateway Interface

Código Fonte: Lib/cgi.py

Módulo de suporte a scripts de Common Gateway Interface (CGI).

Este módulo define vários utilitários para uso por scripts CGI escritos em Python.

22.2.1 Introdução

Um script CGI é chamado por um servidor HTTP, geralmente para processar a entrada do usuário enviada por meio de um elemento HTML `<FORM>` ou `<ISINDEX>`.

Na maioria das vezes, os scripts CGI residem no diretório especial `cgi-bin` do servidor. O servidor HTTP coloca todos os tipos de informações sobre a solicitação (como o nome do host do cliente, a URL solicitada, a string de consulta e muitos outros itens) no ambiente de shell do script, executa o script e envia a saída do script de volta para o cliente.

A entrada do script também está conectada ao cliente e, às vezes, os dados do formulário são lidos dessa forma; em outras ocasiões, os dados do formulário são transmitidos por meio da parte "string de consulta" da URL. Este módulo tem como objetivo cuidar dos diferentes casos e fornecer uma interface mais simples para o script Python. Ele também fornece vários utilitários que ajudam na depuração de scripts, e a última adição é o suporte para uploads de arquivos de um formulário (se o seu navegador permitir).

A saída de um script CGI deve consistir em duas seções, separadas por uma linha em branco. A primeira seção contém vários cabeçalhos, informando ao cliente que tipo de dados está seguindo. O código Python para gerar uma seção de cabeçalho mínima se parece com isto:

```
print("Content-Type: text/html")    # HTML is following
print()                             # blank line, end of headers
```

A segunda seção é geralmente HTML, o que permite que o software cliente exiba um texto bem formatado com cabeçalho, imagens em linha etc. Aqui está o código Python que imprime um pedaço simples de HTML

```
print("<TITLE>CGI script output</TITLE>")
print("<H1>This is my first CGI script</H1>")
print("Hello, world!")
```

22.2.2 Usando o módulo cgi

Comece escrevendo `import cgi`.

Ao escrever um novo script, considere adicionar estas linhas:

```
import cgitb
cgitb.enable()
```

Isso ativa um manipulador de exceção especial que exibirá relatórios detalhados no navegador web se ocorrer algum erro. Se você preferir não mostrar as entranhas de seu programa aos usuários de seu script, você pode salvar os relatórios em arquivos, com um código como este:

```
import cgitb
cgitb.enable(display=0, logdir="/path/to/logdir")
```

É muito útil usar esse recurso durante o desenvolvimento do script. Os relatórios produzidos por *cgitb* fornecem informações que podem economizar muito tempo no rastreamento de bugs. Você sempre pode remover a linha `cgitb` mais tarde, quando tiver testado seu script e estiver confiante de que ele funciona corretamente.

Para obter os dados do formulário enviado, use a classe `FieldStorage`. Se o formulário contiver caracteres não ASCII, use o parâmetro nomeado *encoding* definido para o valor da codificação definida para o documento. Geralmente está contido na tag META na seção HEAD do documento HTML ou pelo cabeçalho *Content-Type*). Isso lê o conteúdo do formulário da entrada padrão ou do ambiente (dependendo do valor de várias variáveis de ambiente definidas de acordo com o padrão CGI). Uma vez que pode consumir a entrada padrão, deve ser instanciado apenas uma vez.

A instância de `FieldStorage` pode ser indexada como um dicionário Python. Ela permite o teste de associação com o operador `in`, e também tem suporte ao método de dicionário padrão *keys()* e a função embutida *len()*. Os campos do formulário contendo strings vazias são ignorados e não aparecem no dicionário; para manter esses valores, forneça um valor verdadeiro para o parâmetro nnomeado opcional *keep_blank_values* ao criar a instância `FieldStorage`.

For instance, the following code (which assumes that the *Content-Type* header and blank line have already been printed) checks that the fields `name` and `addr` are both set to a non-empty string:

```
form = cgi.FieldStorage()
if "name" not in form or "addr" not in form:
    print("<H1>Error</H1>")
    print("Please fill in the name and addr fields.")
    return
print("<p>name:", form["name"].value)
print("<p>addr:", form["addr"].value)
...further form processing here...
```

Here the fields, accessed through `form[key]`, are themselves instances of `FieldStorage` (or `MiniFieldStorage`, depending on the form encoding). The `value` attribute of the instance yields the string value of the field. The `getvalue()` method returns this string value directly; it also accepts an optional second argument as a default to return if the requested key is not present.

If the submitted form data contains more than one field with the same name, the object retrieved by `form[key]` is not a `FieldStorage` or `MiniFieldStorage` instance but a list of such instances. Similarly, in this situation, `form.getvalue(key)` would return a list of strings. If you expect this possibility (when your HTML form contains multiple fields with the same name), use the *getlist()* method, which always returns a list of values (so that you do not need to special-case the single item case). For example, this code concatenates any number of username fields, separated by commas:

```
value = form.getlist("username")
usernames = ",".join(value)
```

If a field represents an uploaded file, accessing the value via the `value` attribute or the `getvalue()` method reads the entire file in memory as bytes. This may not be what you want. You can test for an uploaded file by testing either the `filename` attribute or the `file` attribute. You can then read the data from the `file` attribute before it is automatically closed as part of the garbage collection of the `FieldStorage` instance (the *`read()`* and *`readline()`* methods will return bytes):

```
fileitem = form["userfile"]
if fileitem.file:
    # It's an uploaded file; count lines
    linecount = 0
    while True:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1
```

`FieldStorage` objects also support being used in a `with` statement, which will automatically close them when done.

If an error is encountered when obtaining the contents of an uploaded file (for example, when the user interrupts the form submission by clicking on a Back or Cancel button) the `done` attribute of the object for the field will be set to the value -1.

The file upload draft standard entertains the possibility of uploading multiple files from one field (using a recursive *`multipart/*`* encoding). When this occurs, the item will be a dictionary-like `FieldStorage` item. This can be determined by testing its `type` attribute, which should be *`multipart/form-data`* (or perhaps another MIME type matching *`multipart/*`*). In this case, it can be iterated over recursively just like the top-level form object.

When a form is submitted in the "old" format (as the query string or as a single data part of type *`application/x-www-form-urlencoded`*), the items will actually be instances of the class `MiniFieldStorage`. In this case, the `list`, `file`, and `filename` attributes are always `None`.

A form submitted via POST that also has a query string will contain both `FieldStorage` and `MiniFieldStorage` items.

Alterado na versão 3.4: The `file` attribute is automatically closed upon the garbage collection of the creating `FieldStorage` instance.

Alterado na versão 3.5: Added support for the context management protocol to the `FieldStorage` class.

22.2.3 Interface de nível mais alto

The previous section explains how to read CGI form data using the `FieldStorage` class. This section describes a higher level interface which was added to this class to allow one to do it in a more readable and intuitive way. The interface doesn't make the techniques described in previous sections obsolete — they are still useful to process file uploads efficiently, for example.

The interface consists of two simple methods. Using the methods you can process form data in a generic way, without the need to worry whether only one or more values were posted under one name.

In the previous section, you learned to write following code anytime you expected a user to post more than one value under one name:

```
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

This situation is common for example when a form contains a group of multiple checkboxes with the same name:

```
<input type="checkbox" name="item" value="1" />
<input type="checkbox" name="item" value="2" />
```

In most situations, however, there's only one form control with a particular name in a form and then you expect and need only one value associated with this name. So you write a script containing for example this code:

```
user = form.getvalue("user").upper()
```

The problem with the code is that you should never expect that a client will provide valid input to your scripts. For example, if a curious user appends another `user=foo` pair to the query string, then the script would crash, because in this situation the `getvalue("user")` method call returns a list instead of a string. Calling the *upper()* method on a list is not valid (since lists do not have a method of this name) and results in an *AttributeError* exception.

Therefore, the appropriate way to read form data values was to always use the code which checks whether the obtained value is a single value or a list of values. That's annoying and leads to less readable scripts.

A more convenient approach is to use the methods *getfirst()* and *getlist()* provided by this higher level interface.

`FieldStorage.`**`getfirst`**(*name*, *default=None*)
: This method always returns only one value associated with form field *name*. The method returns only the first value in case that more values were posted under such name. Please note that the order in which the values are received may vary from browser to browser and should not be counted on.[1] If no such form field or value exists then the method returns the value specified by the optional parameter *default*. This parameter defaults to `None` if not specified.

`FieldStorage.`**`getlist`**(*name*)
: This method always returns a list of values associated with form field *name*. The method returns an empty list if no such form field or value exists for *name*. It returns a list consisting of one item if only one such value exists.

Using these methods you can write nice compact code:

```
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", "").upper()    # This way it's safe.
for item in form.getlist("item"):
    do_something(item)
```

22.2.4 Funções

These are useful if you want more control, or if you want to employ some of the algorithms implemented in this module in other circumstances.

`cgi.`**`parse`**(*fp=None*, *environ=os.environ*, *keep_blank_values=False*, *strict_parsing=False*, *separator="&"*)
: Parse a query in the environment or from a file (the file defaults to `sys.stdin`). The *keep_blank_values*, *strict_parsing* and *separator* parameters are passed to *urllib.parse.parse_qs()* unchanged.

`cgi.`**`parse_qs`**(*qs*, *keep_blank_values=False*, *strict_parsing=False*)
: This function is deprecated in this module. Use *urllib.parse.parse_qs()* instead. It is maintained here only for backward compatibility.

`cgi.`**`parse_qsl`**(*qs*, *keep_blank_values=False*, *strict_parsing=False*)
: This function is deprecated in this module. Use *urllib.parse.parse_qsl()* instead. It is maintained here only for backward compatibility.

[1] Note that some recent versions of the HTML specification do state what order the field values should be supplied in, but knowing whether a request was received from a conforming browser, or even from a browser at all, is tedious and error-prone.

cgi.**parse_multipart**(*fp*, *pdict*, *encoding="utf-8"*, *errors="replace"*, *separator="&"*)
Parse input of type *multipart/form-data* (for file uploads). Arguments are *fp* for the input file, *pdict* for a dictionary containing other parameters in the *Content-Type* header, and *encoding*, the request encoding.

Returns a dictionary just like *urllib.parse.parse_qs()*: keys are the field names, each value is a list of values for that field. For non-file fields, the value is a list of strings.

This is easy to use but not much good if you are expecting megabytes to be uploaded — in that case, use the `FieldStorage` class instead which is much more flexible.

Alterado na versão 3.7: Added the *encoding* and *errors* parameters. For non-file fields, the value is now a list of strings, not bytes.

Alterado na versão 3.7.10: Added the *separator* parameter.

cgi.**parse_header**(*string*)
Parse a MIME header (such as *Content-Type*) into a main value and a dictionary of parameters.

cgi.**test**()
Robust test CGI script, usable as main program. Writes minimal HTTP headers and formats all information provided to the script in HTML form.

cgi.**print_environ**()
Format the shell environment in HTML.

cgi.**print_form**(*form*)
Format a form in HTML.

cgi.**print_directory**()
Format the current directory in HTML.

cgi.**print_environ_usage**()
Print a list of useful (used by CGI) environment variables in HTML.

cgi.**escape**(*s*, *quote=False*)
Convert the characters `'&'`, `'<'` and `'>'` in string *s* to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. If the optional flag *quote* is true, the quotation mark character (`"`) is also translated; this helps for inclusion in an HTML attribute value delimited by double quotes, as in `<a href="...">`. Note that single quotes are never translated.

Obsoleto desde a versão 3.2: This function is unsafe because *quote* is false by default, and therefore deprecated. Use *html.escape()* instead.

22.2.5 Caring about security

There's one important rule: if you invoke an external program (via the *os.system()* or *os.popen()* functions. or others with similar functionality), make very sure you don't pass arbitrary strings received from the client to the shell. This is a well-known security hole whereby clever hackers anywhere on the Web can exploit a gullible CGI script to invoke arbitrary shell commands. Even parts of the URL or field names cannot be trusted, since the request doesn't have to come from your form!

To be on the safe side, if you must pass a string gotten from a form to a shell command, you should make sure the string contains only alphanumeric characters, dashes, underscores, and periods.

22.2.6 Installing your CGI script on a Unix system

Read the documentation for your HTTP server and check with your local system administrator to find the directory where CGI scripts should be installed; usually this is in a directory `cgi-bin` in the server tree.

Make sure that your script is readable and executable by "others"; the Unix file mode should be `0o755` octal (use `chmod 0755 filename`). Make sure that the first line of the script contains `#!` starting in column 1 followed by the pathname of the Python interpreter, for instance:

```
#!/usr/local/bin/python
```

Make sure the Python interpreter exists and is executable by "others".

Make sure that any files your script needs to read or write are readable or writable, respectively, by "others" — their mode should be `0o644` for readable and `0o666` for writable. This is because, for security reasons, the HTTP server executes your script as user "nobody", without any special privileges. It can only read (write, execute) files that everybody can read (write, execute). The current directory at execution time is also different (it is usually the server's cgi-bin directory) and the set of environment variables is also different from what you get when you log in. In particular, don't count on the shell's search path for executables (`PATH`) or the Python module search path (`PYTHONPATH`) to be set to anything interesting.

If you need to load modules from a directory which is not on Python's default module search path, you can change the path in your script, before importing other modules. For example:

```
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

(This way, the directory inserted last will be searched first!)

Instructions for non-Unix systems will vary; check your HTTP server's documentation (it will usually have a section on CGI scripts).

22.2.7 Testing your CGI script

Unfortunately, a CGI script will generally not run when you try it from the command line, and a script that works perfectly from the command line may fail mysteriously when run from the server. There's one reason why you should still test your script from the command line: if it contains a syntax error, the Python interpreter won't execute it at all, and the HTTP server will most likely send a cryptic error to the client.

Assuming your script has no syntax errors, yet it does not work, you have no choice but to read the next section.

22.2.8 Debugging CGI scripts

First of all, check for trivial installation errors — reading the section above on installing your CGI script carefully can save you a lot of time. If you wonder whether you have understood the installation procedure correctly, try installing a copy of this module file (`cgi.py`) as a CGI script. When invoked as a script, the file will dump its environment and the contents of the form in HTML form. Give it the right mode etc, and send it a request. If it's installed in the standard `cgi-bin` directory, it should be possible to send it a request by entering a URL into your browser of the form:

```
http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home
```

If this gives an error of type 404, the server cannot find the script – perhaps you need to install it in a different directory. If it gives another error, there's an installation problem that you should fix before trying to go any further. If you get a nicely formatted listing of the environment and form content (in this example, the fields should be listed as "addr" with

value “At Home” and “name” with value “Joe Blow”), the `cgi.py` script has been installed correctly. If you follow the same procedure for your own script, you should now be able to debug it.

The next step could be to call the *cgi* module’s *test()* function from your script: replace its main code with the single statement

```
cgi.test()
```

This should produce the same results as those gotten from installing the `cgi.py` file itself.

When an ordinary Python script raises an unhandled exception (for whatever reason: of a typo in a module name, a file that can’t be opened, etc.), the Python interpreter prints a nice traceback and exits. While the Python interpreter will still do this when your CGI script raises an exception, most likely the traceback will end up in one of the HTTP server’s log files, or be discarded altogether.

Fortunately, once you have managed to get your script to execute *some* code, you can easily send tracebacks to the Web browser using the *cgitb* module. If you haven’t done so already, just add the lines:

```
import cgitb
cgitb.enable()
```

to the top of your script. Then try running it again; when a problem occurs, you should see a detailed report that will likely make apparent the cause of the crash.

If you suspect that there may be a problem in importing the *cgitb* module, you can use an even more robust approach (which only uses built-in modules):

```
import sys
sys.stderr = sys.stdout
print("Content-Type: text/plain")
print()
...your code here...
```

This relies on the Python interpreter to print the traceback. The content type of the output is set to plain text, which disables all HTML processing. If your script works, the raw HTML will be displayed by your client. If it raises an exception, most likely after the first two lines have been printed, a traceback will be displayed. Because no HTML interpretation is going on, the traceback will be readable.

22.2.9 Common problems and solutions

- Most HTTP servers buffer the output from CGI scripts until the script is completed. This means that it is not possible to display a progress report on the client’s display while the script is running.
- Check the installation instructions above.
- Check the HTTP server’s log files. (`tail -f logfile` in a separate window may be useful!)
- Always check a script for syntax errors first, by doing something like `python script.py`.
- If your script does not have any syntax errors, try adding `import cgitb; cgitb.enable()` to the top of the script.
- When invoking external programs, make sure they can be found. Usually, this means using absolute path names — `PATH` is usually not set to a very useful value in a CGI script.
- When reading or writing external files, make sure they can be read or written by the userid under which your CGI script will be running: this is typically the userid under which the web server is running, or some explicitly specified userid for a web server’s `suexec` feature.
- Don’t try to give a CGI script a set-uid mode. This doesn’t work on most systems, and is a security liability as well.

22.3 cgitb — Gerenciador de traceback (situação da pilha de execução) para roteiros de CGI

Código Fonte: Lib/cgitb.py

O módulo *cgitb* fornece um tratador de exceção especial para scripts Python. (Seu nome é um pouco enganador. Ele foi originalmente projetado para exibir informações abrangentes de rastreamento em HTML para scripts CGI. Posteriormente, foi generalizado também para exibir essas informações em texto sem formatação.) Após esse módulo ser ativado, se ocorrer uma exceção não detectada, um relatório detalhado e formatado será exibido. O relatório inclui um traceback mostrando trechos do código-fonte para cada nível, bem como os valores dos argumentos e variáveis locais das funções atualmente em execução, para ajudá-lo a depurar o problema. Opcionalmente, você pode salvar essas informações em um arquivo em vez de enviá-las para o navegador.

Para habilitar esse recurso, basta adicioná-lo ao topo do seu script CGI:

```
import cgitb
cgitb.enable()
```

As opções da função *enable()* controlam se o relatório é exibido no navegador e se o relatório é registrado em um arquivo para análise posterior.

cgitb.**enable**(*display=1*, *logdir=None*, *context=5*, *format="html"*)

Esta função faz com que o módulo *cgitb* assumir o tratamento padrão do interpretador para exceções definindo o valor de *sys.excepthook*.

O argumento opcional *display* é padronizado como `1` e pode ser definido como `0` para suprimir o envio do traceback ao navegador. Se o argumento *logdir* estiver presente, os relatórios de traceback serão gravados nos arquivos. O valor de *logdir* deve ser um diretório em que esses arquivos serão colocados. O argumento opcional *context* é o número de linhas de contexto a serem exibidas em torno da linha atual do código-fonte no traceback; o padrão é `5`. Se o argumento opcional *format* for `"html"`, a saída será formatada como HTML. Qualquer outro valor força a saída de texto sem formatação. O valor padrão é `"html"`.

cgitb.**text**(*info*, *context=5*)

Esta função lida com a exceção descrita por *info* (uma tupla com 3 tuplas contendo o resultado de *sys.exc_info()*), formatando seu retorno como texto e retornando o resultado como uma string. O argumento opcional *context* é o número de linhas de contexto a serem exibidas em torno da linha atual do código-fonte no traceback; o padrão é `5`.

cgitb.**html**(*info*, *context=5*)

Esta função lida com a exceção descrita por *info* (uma tupla com 3 tuplas contendo o resultado de *sys.exc_info()*), formatando seu retorno como HTML e retornando o resultado como uma string. O argumento opcional *context* é o número de linhas de contexto a serem exibidas em torno da linha atual do código-fonte no traceback; o padrão é `5`.

cgitb.**handler**(*info=None*)

Essa função trata uma exceção usando as configurações padrão (ou seja, mostra um relatório no navegador, mas não faz logon em um arquivo). Isso pode ser usado quando você capturou uma exceção e deseja denunciá-la usando *cgitb*. O argumento opcional *info* deve ser uma tupla de três, contendo um tipo de exceção, um valor de exceção e um objeto de traceback exatamente como a tupla retornada por *sys.exc_info()*. Se o argumento *info* não for fornecido, a exceção atual será obtida em *sys.exc_info()*.

22.4 `wsgiref` — Utilidades WSGI e Implementação de Referência

A Interface de Gateway para Servidor Web (em inglês, Web Server Gateway Interface - WSGI) é uma interface padrão localizada entre software de servidores web e aplicações web escritas em Python. Ter um interface padrão torna mais fácil a utilização de uma aplicação que suporta WSGI com inúmeros diferentes servidores web.

Apenas autores de servidores web e frameworks de programação necessitam saber todos os detalhes e especificidades do design WSGI. Você não precisa entender todos os detalhes do WSGI para apenas instalar uma aplicação WSGI ou para escrever uma aplicação web usando um framework existente.

wsgiref é uma implementação de referência da especificação WSGI que pode ser utilizada para adicionar suporte WSGI em um servidor web ou framework. Ele provê utilidades para manipulação de variáveis de ambiente WSGI e cabeçalhos de respostas, classes base para implementação de servidores WSGI, uma demo de um servidor HTTP que serve aplicações WSGI e uma ferramenta de validação que confere se servidores WSGI e aplicações estão de acordo com a especificação WSGI (**PEP 3333**).

Veja ‘wsgi.readthedocs.io <https://wsgi.readthedocs.io/>‘_para mais informações sobre WSGI, além de links para tutoriais e outros recursos.

22.4.1 `wsgiref.util` – Utilidades do ambiente WSGI

This module provides a variety of utility functions for working with WSGI environments. A WSGI environment is a dictionary containing HTTP request variables as described in **PEP 3333**. All of the functions taking an *environ* parameter expect a WSGI-compliant dictionary to be supplied; please see **PEP 3333** for a detailed specification.

`wsgiref.util.`**`guess_scheme`**(*environ*)

Retorna uma sugestão sobre “wsgi.url_scheme“ser “http” ou “https” buscando por uma “HTTPS“variável de ambiente dentro do dicionário *environ*. O valor de retorno é uma string.

This function is useful when creating a gateway that wraps CGI or a CGI-like protocol such as FastCGI. Typically, servers providing such protocols will include a `HTTPS` variable with a value of “1” “yes”, or “on” when a request is received via SSL. So, this function returns “https” if such a value is found, and “http” otherwise.

`wsgiref.util.`**`request_uri`**(*environ*, *include_query=True*)

Return the full request URI, optionally including the query string, using the algorithm found in the “URL Reconstruction” section of **PEP 3333**. If *include_query* is false, the query string is not included in the resulting URI.

`wsgiref.util.`**`application_uri`**(*environ*)

Similar to *`request_uri()`*, except that the `PATH_INFO` and `QUERY_STRING` variables are ignored. The result is the base URI of the application object addressed by the request.

`wsgiref.util.`**`shift_path_info`**(*environ*)

Shift a single name from `PATH_INFO` to `SCRIPT_NAME` and return the name. The *environ* dictionary is *modified* in-place; use a copy if you need to keep the original `PATH_INFO` or `SCRIPT_NAME` intact.

If there are no remaining path segments in `PATH_INFO`, `None` is returned.

Typically, this routine is used to process each portion of a request URI path, for example to treat the path as a series of dictionary keys. This routine modifies the passed-in environment to make it suitable for invoking another WSGI application that is located at the target URI. For example, if there is a WSGI application at `/foo`, and the request URI path is `/foo/bar/baz`, and the WSGI application at `/foo` calls *`shift_path_info()`*, it will receive the string “bar”, and the environment will be updated to be suitable for passing to a WSGI application at `/foo/bar`. That is, `SCRIPT_NAME` will change from `/foo` to `/foo/bar`, and `PATH_INFO` will change from `/bar/baz` to `/baz`.

When PATH_INFO is just a "/", this routine returns an empty string and appends a trailing slash to SCRIPT_NAME, even though empty path segments are normally ignored, and SCRIPT_NAME doesn't normally end in a slash. This is intentional behavior, to ensure that an application can tell the difference between URIs ending in /x from ones ending in /x/ when using this routine to do object traversal.

wsgiref.util.**setup_testing_defaults**(*environ*)
Update *environ* with trivial defaults for testing purposes.

This routine adds various parameters required for WSGI, including HTTP_HOST, SERVER_NAME, SERVER_PORT, REQUEST_METHOD, SCRIPT_NAME, PATH_INFO, and all of the **PEP 3333**-defined wsgi.* variables. It only supplies default values, and does not replace any existing settings for these variables.

This routine is intended to make it easier for unit tests of WSGI servers and applications to set up dummy environments. It should NOT be used by actual WSGI servers or applications, since the data is fake!

Exemplo de uso:

```
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It's going to print out the
# environment dictionary after being updated by setup_testing_defaults
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain; charset=utf-8')]

    start_response(status, headers)

    ret = [("%s: %s\n" % (key, value)).encode("utf-8")
           for key, value in environ.items()]
    return ret

with make_server('', 8000, simple_app) as httpd:
    print("Serving on port 8000...")
    httpd.serve_forever()
```

In addition to the environment functions above, the *wsgiref.util* module also provides these miscellaneous utilities:

wsgiref.util.**is_hop_by_hop**(*header_name*)
Return True if 'header_name' is an HTTP/1.1 "Hop-by-Hop" header, as defined by **RFC 2616**.

class wsgiref.util.**FileWrapper**(*filelike*, *blksize=8192*)
A wrapper to convert a file-like object to an *iterator*. The resulting objects support both __getitem__() and __iter__() iteration styles, for compatibility with Python 2.1 and Jython. As the object is iterated over, the optional *blksize* parameter will be repeatedly passed to the *filelike* object's read() method to obtain bytestrings to yield. When read() returns an empty bytestring, iteration is ended and is not resumable.

If *filelike* has a close() method, the returned object will also have a close() method, and it will invoke the *filelike* object's close() method when called.

Exemplo de uso:

```
from io import StringIO
from wsgiref.util import FileWrapper

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object"*10)
```

(continua na próxima página)

(continuação da página anterior)

```
wrapper = FileWrapper(filelike, blksize=5)

for chunk in wrapper:
    print(chunk)
```

22.4.2 wsgiref.headers – WSGI response header tools

This module provides a single class, *Headers*, for convenient manipulation of WSGI response headers using a mapping-like interface.

class wsgiref.headers.**Headers**([*headers*])

Create a mapping-like object wrapping *headers*, which must be a list of header name/value tuples as described in **PEP 3333**. The default value of *headers* is an empty list.

Headers objects support typical mapping operations including `__getitem__()`, `get()`, `__setitem__()`, `setdefault()`, `__delitem__()` and `__contains__()`. For each of these methods, the key is the header name (treated case-insensitively), and the value is the first value associated with that header name. Setting a header deletes any existing values for that header, then adds a new value at the end of the wrapped header list. Headers' existing order is generally maintained, with new headers added to the end of the wrapped list.

Unlike a dictionary, *Headers* objects do not raise an error when you try to get or delete a key that isn't in the wrapped header list. Getting a nonexistent header just returns `None`, and deleting a nonexistent header does nothing.

Headers objects also support `keys()`, `values()`, and `items()` methods. The lists returned by `keys()` and `items()` can include the same key more than once if there is a multi-valued header. The `len()` of a *Headers* object is the same as the length of its `items()`, which is the same as the length of the wrapped header list. In fact, the `items()` method just returns a copy of the wrapped header list.

Calling `bytes()` on a *Headers* object returns a formatted bytestring suitable for transmission as HTTP response headers. Each header is placed on a line with its value, separated by a colon and a space. Each line is terminated by a carriage return and line feed, and the bytestring is terminated with a blank line.

In addition to their mapping interface and formatting features, *Headers* objects also have the following methods for querying and adding multi-valued headers, and for adding headers with MIME parameters:

get_all(*name*)

Return a list of all the values for the named header.

The returned list will be sorted in the order they appeared in the original header list or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list. If no fields exist with the given name, returns an empty list.

add_header(*name*, *value*, ***_params*)

Add a (possibly multi-valued) header, with optional MIME parameters specified via keyword arguments.

name is the header field to add. Keyword arguments can be used to set MIME parameters for the header field. Each parameter must be a string or `None`. Underscores in parameter names are converted to dashes, since dashes are illegal in Python identifiers, but many MIME parameter names include dashes. If the parameter value is a string, it is added to the header value parameters in the form `name="value"`. If it is `None`, only the parameter name is added. (This is used for MIME parameters without a value.) Example usage:

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

The above will add a header that looks like this:

```
Content-Disposition: attachment; filename="bud.gif"
```

Alterado na versão 3.5: o parâmetro *headers* é opcional.

22.4.3 `wsgiref.simple_server` – a simple WSGI HTTP server

This module implements a simple HTTP server (based on `http.server`) that serves WSGI applications. Each server instance serves a single WSGI application on a given host and port. If you want to serve multiple applications on a single host and port, you should create a WSGI application that parses `PATH_INFO` to select which application to invoke for each request. (E.g., using the `shift_path_info()` function from `wsgiref.util`.)

`wsgiref.simple_server.`**`make_server`**(*host*, *port*, *app*, *server_class=WSGIServer*, *handler_class=WSGIRequestHandler*)

Create a new WSGI server listening on *host* and *port*, accepting connections for *app*. The return value is an instance of the supplied *server_class*, and will process requests using the specified *handler_class*. *app* must be a WSGI application object, as defined by **PEP 3333**.

Exemplo de uso:

```
from wsgiref.simple_server import make_server, demo_app

with make_server('', 8000, demo_app) as httpd:
    print("Serving HTTP on port 8000...")

    # Respond to requests until process is killed
    httpd.serve_forever()

    # Alternative: serve one request, then exit
    httpd.handle_request()
```

`wsgiref.simple_server.`**`demo_app`**(*environ*, *start_response*)

This function is a small but complete WSGI application that returns a text page containing the message "Hello world!" and a list of the key/value pairs provided in the *environ* parameter. It's useful for verifying that a WSGI server (such as `wsgiref.simple_server`) is able to run a simple WSGI application correctly.

`class` `wsgiref.simple_server.`**`WSGIServer`**(*server_address*, *RequestHandlerClass*)

Create a `WSGIServer` instance. *server_address* should be a `(host,port)` tuple, and *RequestHandlerClass* should be the subclass of `http.server.BaseHTTPRequestHandler` that will be used to process requests.

You do not normally need to call this constructor, as the `make_server()` function can handle all the details for you.

`WSGIServer` is a subclass of `http.server.HTTPServer`, so all of its methods (such as `serve_forever()` and `handle_request()`) are available. `WSGIServer` also provides these WSGI-specific methods:

`set_app`(*application*)

Sets the callable *application* as the WSGI application that will receive requests.

`get_app`()

Returns the currently-set application callable.

Normally, however, you do not need to use these additional methods, as `set_app()` is normally called by `make_server()`, and the `get_app()` exists mainly for the benefit of request handler instances.

`class` `wsgiref.simple_server.`**`WSGIRequestHandler`**(*request*, *client_address*, *server*)

Create an HTTP handler for the given *request* (i.e. a socket), *client_address* (a `(host,port)` tuple), and *server* (`WSGIServer` instance).

You do not need to create instances of this class directly; they are automatically created as needed by *WSGIServer* objects. You can, however, subclass this class and supply it as a *handler_class* to the *make_server()* function. Some possibly relevant methods for overriding in subclasses:

get_environ()
Returns a dictionary containing the WSGI environment for a request. The default implementation copies the contents of the *WSGIServer* object's `base_environ` dictionary attribute and then adds various headers derived from the HTTP request. Each call to this method should return a new dictionary containing all of the relevant CGI environment variables as specified in **PEP 3333**.

get_stderr()
Return the object that should be used as the `wsgi.errors` stream. The default implementation just returns `sys.stderr`.

handle()
Process the HTTP request. The default implementation creates a handler instance using a *wsgiref.handlers* class to implement the actual WSGI application interface.

22.4.4 `wsgiref.validate` — WSGI conformance checker

When creating new WSGI application objects, frameworks, servers, or middleware, it can be useful to validate the new code's conformance using *wsgiref.validate*. This module provides a function that creates WSGI application objects that validate communications between a WSGI server or gateway and a WSGI application object, to check both sides for protocol conformance.

Note that this utility does not guarantee complete **PEP 3333** compliance; an absence of errors from this module does not necessarily mean that errors do not exist. However, if this module does produce an error, then it is virtually certain that either the server or application is not 100% compliant.

This module is based on the `paste.lint` module from Ian Bicking's "Python Paste" library.

`wsgiref.validate.`**validator**(*application*)
Wrap *application* and return a new WSGI application object. The returned application will forward all requests to the original *application*, and will check that both the *application* and the server invoking it are conforming to the WSGI specification and to **RFC 2616**.

Any detected nonconformance results in an *AssertionError* being raised; note, however, that how these errors are handled is server-dependent. For example, *wsgiref.simple_server* and other servers based on *wsgiref.handlers* (that don't override the error handling methods to do something else) will simply output a message that an error has occurred, and dump the traceback to `sys.stderr` or some other error stream.

This wrapper may also generate output using the *warnings* module to indicate behaviors that are questionable but which may not actually be prohibited by **PEP 3333**. Unless they are suppressed using Python command-line options or the *warnings* API, any such warnings will be written to `sys.stderr` (*not* `wsgi.errors`, unless they happen to be the same object).

Exemplo de uso:

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not compliant to the
# standard, so the validator is going to break
def simple_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

```

(continua na próxima página)

(continuação da página anterior)

```
    # This is going to break because we need to return a list, and
    # the validator is going to inform us
    return b"Hello World"

# This is the application wrapped in a validator
validator_app = validator(simple_app)

with make_server('', 8000, validator_app) as httpd:
    print("Listening on port 8000....")
    httpd.serve_forever()
```

22.4.5 `wsgiref.handlers` – server/gateway base classes

This module provides base handler classes for implementing WSGI servers and gateways. These base classes handle most of the work of communicating with a WSGI application, as long as they are given a CGI-like environment, along with input, output, and error streams.

class wsgiref.handlers.**CGIHandler**

CGI-based invocation via `sys.stdin`, `sys.stdout`, `sys.stderr` and `os.environ`. This is useful when you have a WSGI application and want to run it as a CGI script. Simply invoke `CGIHandler().run(app)`, where `app` is the WSGI application object you wish to invoke.

This class is a subclass of *BaseCGIHandler* that sets `wsgi.run_once` to true, `wsgi.multithread` to false, and `wsgi.multiprocess` to true, and always uses *sys* and *os* to obtain the necessary CGI streams and environment.

class wsgiref.handlers.**IISCGIHandler**

A specialized alternative to *CGIHandler*, for use when deploying on Microsoft's IIS web server, without having set the config allowPathInfo option (IIS>=7) or metabase allowPathInfoForScriptMappings (IIS<7).

By default, IIS gives a `PATH_INFO` that duplicates the `SCRIPT_NAME` at the front, causing problems for WSGI applications that wish to implement routing. This handler strips any such duplicated path.

IIS can be configured to pass the correct `PATH_INFO`, but this causes another bug where `PATH_TRANSLATED` is wrong. Luckily this variable is rarely used and is not guaranteed by WSGI. On IIS<7, though, the setting can only be made on a vhost level, affecting all other script mappings, many of which break when exposed to the `PATH_TRANSLATED` bug. For this reason IIS<7 is almost never deployed with the fix. (Even IIS7 rarely uses it because there is still no UI for it.)

There is no way for CGI code to tell whether the option was set, so a separate handler class is provided. It is used in the same way as *CGIHandler*, i.e., by calling `IISCGIHandler().run(app)`, where `app` is the WSGI application object you wish to invoke.

Novo na versão 3.2.

class wsgiref.handlers.**BaseCGIHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

Similar to *CGIHandler*, but instead of using the *sys* and *os* modules, the CGI environment and I/O streams are specified explicitly. The *multithread* and *multiprocess* values are used to set the `wsgi.multithread` and `wsgi.multiprocess` flags for any applications run by the handler instance.

This class is a subclass of *SimpleHandler* intended for use with software other than HTTP "origin servers". If you are writing a gateway protocol implementation (such as CGI, FastCGI, SCGI, etc.) that uses a `Status:` header to send an HTTP status, you probably want to subclass this instead of *SimpleHandler*.

class wsgiref.handlers.**SimpleHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

Similar to *BaseCGIHandler*, but designed for use with HTTP origin servers. If you are writing an HTTP server implementation, you will probably want to subclass this instead of *BaseCGIHandler*.

This class is a subclass of *BaseHandler*. It overrides the `__init__()`, `get_stdin()`, `get_stderr()`, `add_cgi_vars()`, `_write()`, and `_flush()` methods to support explicitly setting the environment and streams via the constructor. The supplied environment and streams are stored in the `stdin`, `stdout`, `stderr`, and `environ` attributes.

The *write()* method of *stdout* should write each chunk in full, like *io.BufferedIOBase*.

class wsgiref.handlers.**BaseHandler**

This is an abstract base class for running WSGI applications. Each instance will handle a single HTTP request, although in principle you could create a subclass that was reusable for multiple requests.

BaseHandler instances have only one method intended for external use:

run(*app*)

Run the specified WSGI application, *app*.

All of the other *BaseHandler* methods are invoked by this method in the process of running the application, and thus exist primarily to allow customizing the process.

The following methods MUST be overridden in a subclass:

_write(*data*)

Buffer the bytes *data* for transmission to the client. It's okay if this method actually transmits the data; *BaseHandler* just separates write and flush operations for greater efficiency when the underlying system actually has such a distinction.

_flush()

Force buffered data to be transmitted to the client. It's okay if this method is a no-op (i.e., if *_write()* actually sends the data).

get_stdin()

Return an input stream object suitable for use as the `wsgi.input` of the request currently being processed.

get_stderr()

Return an output stream object suitable for use as the `wsgi.errors` of the request currently being processed.

add_cgi_vars()

Insert CGI variables for the current request into the `environ` attribute.

Here are some other methods and attributes you may wish to override. This list is only a summary, however, and does not include every method that can be overridden. You should consult the docstrings and source code for additional information before attempting to create a customized *BaseHandler* subclass.

Attributes and methods for customizing the WSGI environment:

wsgi_multithread

The value to be used for the `wsgi.multithread` environment variable. It defaults to true in *BaseHandler*, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_multiprocess

The value to be used for the `wsgi.multiprocess` environment variable. It defaults to true in *BaseHandler*, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_run_once

The value to be used for the `wsgi.run_once` environment variable. It defaults to false in *BaseHandler*, but *CGIHandler* sets it to true by default.

os_environ

The default environment variables to be included in every request's WSGI environment. By default, this is

a copy of `os.environ` at the time that *wsgiref.handlers* was imported, but subclasses can either create their own at the class or instance level. Note that the dictionary should be considered read-only, since the default value is shared between multiple classes and instances.

server_software
If the *origin_server* attribute is set, this attribute's value is used to set the default `SERVER_SOFTWARE` WSGI environment variable, and also to set a default `Server:` header in HTTP responses. It is ignored for handlers (such as *BaseCGIHandler* and *CGIHandler*) that are not HTTP origin servers.

Alterado na versão 3.3: The term "Python" is replaced with implementation specific term like "CPython", "Jython" etc.

get_scheme()
Return the URL scheme being used for the current request. The default implementation uses the `guess_scheme()` function from *wsgiref.util* to guess whether the scheme should be "http" or "https", based on the current request's `environ` variables.

setup_environ()
Set the `environ` attribute to a fully-populated WSGI environment. The default implementation uses all of the above methods and attributes, plus the *get_stdin()*, *get_stderr()*, and *add_cgi_vars()* methods and the *wsgi_file_wrapper* attribute. It also inserts a `SERVER_SOFTWARE` key if not present, as long as the *origin_server* attribute is a true value and the *server_software* attribute is set.

Methods and attributes for customizing exception handling:

log_exception(*exc_info*)
Log the *exc_info* tuple in the server log. *exc_info* is a `(type, value, traceback)` tuple. The default implementation simply writes the traceback to the request's `wsgi.errors` stream and flushes it. Subclasses can override this method to change the format or retarget the output, mail the traceback to an administrator, or whatever other action may be deemed suitable.

traceback_limit
The maximum number of frames to include in tracebacks output by the default *log_exception()* method. If `None`, all frames are included.

error_output(*environ*, *start_response*)
This method is a WSGI application to generate an error page for the user. It is only invoked if an error occurs before headers are sent to the client.

This method can access the current error information using `sys.exc_info()`, and should pass that information to *start_response* when calling it (as described in the "Error Handling" section of **PEP 3333**).

The default implementation just uses the *error_status*, *error_headers*, and *error_body* attributes to generate an output page. Subclasses can override this to produce more dynamic error output.

Note, however, that it's not recommended from a security perspective to spit out diagnostics to any old user; ideally, you should have to do something special to enable diagnostic output, which is why the default implementation doesn't include any.

error_status
The HTTP status used for error responses. This should be a status string as defined in **PEP 3333**; it defaults to a 500 code and message.

error_headers
The HTTP headers used for error responses. This should be a list of WSGI response headers (`(name, value)` tuples), as described in **PEP 3333**. The default list just sets the content type to `text/plain`.

error_body
The error response body. This should be an HTTP response body bytestring. It defaults to the plain text, "A server error occurred. Please contact the administrator."

Methods and attributes for **PEP 3333**'s "Optional Platform-Specific File Handling" feature:

wsgi_file_wrapper
A `wsgi.file_wrapper` factory, or `None`. The default value of this attribute is the *wsgiref.util.FileWrapper* class.

sendfile()
Override to implement platform-specific file transmission. This method is called only if the application's return value is an instance of the class specified by the *wsgi_file_wrapper* attribute. It should return a true value if it was able to successfully transmit the file, so that the default transmission code will not be executed. The default implementation of this method just returns a false value.

Miscellaneous methods and attributes:

origin_server
This attribute should be set to a true value if the handler's *_write()* and *_flush()* are being used to communicate directly to the client, rather than via a CGI-like gateway protocol that wants the HTTP status in a special `Status:` header.

This attribute's default value is true in *BaseHandler*, but false in *BaseCGIHandler* and *CGIHandler*.

http_version
If *origin_server* is true, this string attribute is used to set the HTTP version of the response set to the client. It defaults to `"1.0"`.

wsgiref.handlers.**read_environ**()
Transcode CGI variables from `os.environ` to PEP 3333 "bytes in unicode" strings, returning a new dictionary. This function is used by *CGIHandler* and *IISCGIHandler* in place of directly using `os.environ`, which is not necessarily WSGI-compliant on all platforms and web servers using Python 3 – specifically, ones where the OS's actual environment is Unicode (i.e. Windows), or ones where the environment is bytes, but the system encoding used by Python to decode it is anything other than ISO-8859-1 (e.g. Unix systems using UTF-8).

If you are implementing a CGI-based handler of your own, you probably want to use this routine instead of just copying values out of `os.environ` directly.

Novo na versão 3.2.

22.4.6 Exemplos

This is a working "Hello World" WSGI application:

```
from wsgiref.simple_server import make_server

# Every WSGI application must have an application object - a callable
# object that accepts two arguments. For that purpose, we're going to
# use a function (note that you're not limited to a function, you can
# use a class for example). The first argument passed to the function
# is a dictionary containing CGI-style environment variables and the
# second variable is the callable object (see PEP 333).
def hello_world_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain; charset=utf-8')]  # HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed
    return [b"Hello World"]

with make_server('', 8000, hello_world_app) as httpd:
```

(continua na próxima página)

(continuação da página anterior)

```
    print("Serving on port 8000...")

    # Serve until process is killed
    httpd.serve_forever()
```

22.5 `urllib` — Módulos de manipulação de URL

Código-fonte: Lib/urllib/

`urllib` é um pacote que coleciona vários módulos para trabalhar com URLs:

- *`urllib.request`* para abrir e ler URLs
- *`urllib.error`* contendo as exceções levantadas por *`urllib.request`*
- *`urllib.parse`* para analisar URLs
- *`urllib.robotparser`* para analisar arquivos `robots.txt`

22.6 `urllib.request` — Biblioteca extensível para abrir URLs

Código Fonte: Lib/urllib/request.py

O módulo :mod: *urllib.request* define funções e classes que ajudam a abrir URLs (principalmente HTTP) em um mundo complexo - autenticação básica ou por digest, redirecionamentos, cookies e muito mais.

Ver também:

O Pacote de requisições é recomendado para uma interface alto-nível de cliente HTTP.

O módulo: mod: *urllib.request* define as seguintes funções:

`urllib.request.`**`urlopen`**(*url*, *data=None*[, *timeout*], *, *cafile=None*, *capath=None*, *cadefault=False*, *context=None*)

Abra o URL * url *, que pode ser uma string ou um objeto: class: *Request.*

data must be an object specifying additional data to be sent to the server, or `None` if no such data is needed. See *`Request`* for details.

O módulo urllib.request usa HTTP/1.1 e inclui o cabeçalho `Connection:close` em suas solicitações HTTP.

O parâmetro opcional *timeout* especifica um tempo limite em segundos para bloquear operações como a tentativa de conexão (se não for especificado, a configuração de tempo limite padrão global será usada). Na verdade, isso só funciona para conexões HTTP, HTTPS e FTP.

Se *context* for especificado, deve ser uma instância de *`ssl.SSLContext`* descrevendo as várias opções SSL. Veja class:~*http.client.HTTPSConnection* para mais detalhes.

Os parâmetros opcionais *cafile* e *capath* especificam um conjunto de certificados de AC confiáveis para solicitações HTTPS. *cafile* deve apontar para um único arquivo contendo um pacote de certificados de AC, enquanto *capath* deve apontar para um diretório de arquivos de certificado em hash. Mais informações podem ser encontradas em *`ssl.SSLContext.load_verify_locations()`*.

O parâmetro *cadefault* é ignorado.

This function always returns an object which can work as a *context manager* and has methods such as

- `geturl()` — return the URL of the resource retrieved, commonly used to determine if a redirect was followed
- `info()` — return the meta-information of the page, such as headers, in the form of an *email.message_from_string()* instance (see Quick Reference to HTTP Headers)
- `getcode()` – retorna o código de status HTTP da resposta.

Para URLs HTTP e HTTPS, esta função retorna um objeto *http.client.HTTPResponse* ligeiramente modificado. Além dos três novos métodos acima, o atributo msg contém as mesmas informações que o atributo *reason* — a frase de razão retornada pelo servidor — em vez dos cabeçalhos de resposta como é especificado na documentação para *HTTPResponse*.

Para FTP, arquivo e URLs de dados e solicitações explicitamente tratadas pelas classes legadas *URLopener* e *FancyURLopener*, esta função retorna um objeto `urllib.response.addinfourl`.

Levanta *URLError* quando ocorrer erros de protocolo.

Observe que `None` pode ser retornado se nenhum manipulador lidar com a solicitação (embora a *OpenerDirector* global instalada padrão use *UnknownHandler* para garantir que isso nunca aconteça).

Além disso, se as configurações de proxy forem detectadas (por exemplo, quando uma variável de ambiente `*_proxy` como `http_proxy` é definida), *ProxyHandler* é instalado por padrão e garante que as solicitações sejam tratadas por meio o proxy.

A função legada `urllib.urlopen` do Python 2.6 e anteriores foi descontinuada; *urllib.request.urlopen()* corresponde à antiga `urllib2.urlopen`. O tratamento de proxy, que foi feito passando um parâmetro de dicionário para `urllib.urlopen`, pode ser obtido usando objetos *ProxyHandler*.

Alterado na versão 3.2: *cafile* e *capath* foram adicionados.

Alterado na versão 3.2: Hosts virtuais HTTPS agora são suportados, se possível (ou seja, se *ssl.HAS_SNI* for verdadeiro).

Novo na versão 3.2: *data* pode ser um oobjeto iterável.

Alterado na versão 3.3: *cadefault* foi adicionado.

Alterado na versão 3.4.3: *context* foi adicionado.

Obsoleto desde a versão 3.6: *cafile*, *capath* e *cadefault* estão descontinuados em favor de *context*. Por favor, em vez disso, utilize *ssl.SSLContext.load_cert_chain()* ou deixe *ssl.create_default_context()* selecionar os certificados AC confiáveis do sistema para você.

`urllib.request.`**`install_opener`**(*opener*)

Install an *OpenerDirector* instance as the default global opener. Installing an opener is only necessary if you want urlopen to use that opener; otherwise, simply call *OpenerDirector.open()* instead of *urlopen()*. The code does not check for a real *OpenerDirector*, and any class with the appropriate interface will work.

`urllib.request.`**`build_opener`**([*handler*, ...])

Return an *OpenerDirector* instance, which chains the handlers in the order given. *handler*s can be either instances of *BaseHandler*, or subclasses of *BaseHandler* (in which case it must be possible to call the constructor without any parameters). Instances of the following classes will be in front of the *handler*s, unless the *handler*s contain them, instances of them or subclasses of them: *ProxyHandler* (if proxy settings are detected), *UnknownHandler*, *HTTPHandler*, *HTTPDefaultErrorHandler*, *HTTPRedirectHandler*, *FTPHandler*, *FileHandler*, *HTTPErrorProcessor*.

If the Python installation has SSL support (i.e., if the *ssl* module can be imported), *HTTPSHandler* will also be added.

A *BaseHandler* subclass may also change its `handler_order` attribute to modify its position in the handlers list.

`urllib.request.`**`pathname2url`**`(`*path*`)`
Convert the pathname *path* from the local syntax for a path to the form used in the path component of a URL. This does not produce a complete URL. The return value will already be quoted using the *quote()* function.

`urllib.request.`**`url2pathname`**`(`*path*`)`
Convert the path component *path* from a percent-encoded URL to the local syntax for a path. This does not accept a complete URL. This function uses *unquote()* to decode *path*.

`urllib.request.`**`getproxies`**`()`
This helper function returns a dictionary of scheme to proxy server URL mappings. It scans the environment for variables named `<scheme>_proxy`, in a case insensitive approach, for all operating systems first, and when it cannot find it, looks for proxy information from Mac OSX System Configuration for Mac OS X and Windows Systems Registry for Windows. If both lowercase and uppercase environment variables exist (and disagree), lowercase is preferred.

Nota: If the environment variable `REQUEST_METHOD` is set, which usually indicates your script is running in a CGI environment, the environment variable `HTTP_PROXY` (uppercase `_PROXY`) will be ignored. This is because that variable can be injected by a client using the "Proxy:" HTTP header. If you need to use an HTTP proxy in a CGI environment, either use `ProxyHandler` explicitly, or make sure the variable name is in lowercase (or at least the `_proxy` suffix).

The following classes are provided:

`class` `urllib.request.`**`Request`**`(`*url*, *data=None*, *headers={}*, *origin_req_host=None*, *unverifiable=False*, *method=None*`)`
This class is an abstraction of a URL request.

url should be a string containing a valid URL.

data must be an object specifying additional data to send to the server, or `None` if no such data is needed. Currently HTTP requests are the only ones that use *data*. The supported object types include bytes, file-like objects, and iterables. If no `Content-Length` nor `Transfer-Encoding` header field has been provided, *HTTPHandler* will set these headers according to the type of *data*. `Content-Length` will be used to send bytes objects, while `Transfer-Encoding: chunked` as specified in **RFC 7230**, Section 3.3.1 will be used to send files and other iterables.

For an HTTP POST request method, *data* should be a buffer in the standard *application/x-www-form-urlencoded* format. The *urllib.parse.urlencode()* function takes a mapping or sequence of 2-tuples and returns an ASCII string in this format. It should be encoded to bytes before being used as the *data* parameter.

headers should be a dictionary, and will be treated as if *add_header()* was called with each key and value as arguments. This is often used to "spoof" the `User-Agent` header value, which is used by a browser to identify itself – some HTTP servers only allow requests coming from common browsers as opposed to scripts. For example, Mozilla Firefox may identify itself as `"Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11"`, while *urllib*'s default user agent string is `"Python-urllib/2.6"` (on Python 2.6).

An appropriate `Content-Type` header should be included if the *data* argument is present. If this header has not been provided and *data* is not None, `Content-Type: application/x-www-form-urlencoded` will be added as a default.

The next two arguments are only of interest for correct handling of third-party HTTP cookies:

origin_req_host should be the request-host of the origin transaction, as defined by **RFC 2965**. It defaults to `http.cookiejar.request_host(self)`. This is the host name or IP address of the original request that was

initiated by the user. For example, if the request is for an image in an HTML document, this should be the request-host of the request for the page containing the image.

unverifiable should indicate whether the request is unverifiable, as defined by **RFC 2965**. It defaults to `False`. An unverifiable request is one whose URL the user did not have the option to approve. For example, if the request is for an image in an HTML document, and the user had no option to approve the automatic fetching of the image, this should be true.

method should be a string that indicates the HTTP request method that will be used (e.g. `'HEAD'`). If provided, its value is stored in the *method* attribute and is used by *get_method()*. The default is `'GET'` if *data* is `None` or `'POST'` otherwise. Subclasses may indicate a different default method by setting the *method* attribute in the class itself.

Nota: The request will not work as expected if the data object is unable to deliver its content more than once (e.g. a file or an iterable that can produce the content only once) and the request is retried for HTTP redirects or authentication. The *data* is sent to the HTTP server right away after the headers. There is no support for a 100-continue expectation in the library.

Alterado na versão 3.3: *Request.method* argument is added to the Request class.

Alterado na versão 3.4: Default *Request.method* may be indicated at the class level.

Alterado na versão 3.6: Do not raise an error if the `Content-Length` has not been provided and *data* is neither `None` nor a bytes object. Fall back to use chunked transfer encoding instead.

class urllib.request.**OpenerDirector**
The *OpenerDirector* class opens URLs via *BaseHandler*s chained together. It manages the chaining of handlers, and recovery from errors.

class urllib.request.**BaseHandler**
This is the base class for all registered handlers — and handles only the simple mechanics of registration.

class urllib.request.**HTTPDefaultErrorHandler**
A class which defines a default handler for HTTP error responses; all responses are turned into *HTTPError* exceptions.

class urllib.request.**HTTPRedirectHandler**
A class to handle redirections.

class urllib.request.**HTTPCookieProcessor**(*cookiejar=None*)
A class to handle HTTP Cookies.

class urllib.request.**ProxyHandler**(*proxies=None*)
Cause requests to go through a proxy. If *proxies* is given, it must be a dictionary mapping protocol names to URLs of proxies. The default is to read the list of proxies from the environment variables `<protocol>_proxy`. If no proxy environment variables are set, then in a Windows environment proxy settings are obtained from the registry's Internet Settings section, and in a Mac OS X environment proxy information is retrieved from the OS X System Configuration Framework.

To disable autodetected proxy pass an empty dictionary.

The `no_proxy` environment variable can be used to specify hosts which shouldn't be reached via proxy; if set, it should be a comma-separated list of hostname suffixes, optionally with `:port` appended, for example `cern.ch,ncsa.uiuc.edu,some.host:8080`.

Nota: `HTTP_PROXY` will be ignored if a variable `REQUEST_METHOD` is set; see the documentation on *getproxies()*.

---

class urllib.request.**HTTPPasswordMgr**
Keep a database of `(realm, uri) -> (user, password)` mappings.

class urllib.request.**HTTPPasswordMgrWithDefaultRealm**
Keep a database of `(realm, uri) -> (user, password)` mappings. A realm of `None` is considered a catch-all realm, which is searched if no other realm fits.

class urllib.request.**HTTPPasswordMgrWithPriorAuth**
A variant of *HTTPPasswordMgrWithDefaultRealm* that also has a database of `uri -> is_authenticated` mappings. Can be used by a BasicAuth handler to determine when to send authentication credentials immediately instead of waiting for a `401` response first.

Novo na versão 3.5.

class urllib.request.**AbstractBasicAuthHandler**(*password_mgr=None*)
This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported. If *passwd_mgr* also provides `is_authenticated` and `update_authenticated` methods (see *HTTPPasswordMgrWithPriorAuth Objects*), then the handler will use the `is_authenticated` result for a given URI to determine whether or not to send authentication credentials with the request. If `is_authenticated` returns `True` for the URI, credentials are sent. If `is_authenticated` is `False`, credentials are not sent, and then if a `401` response is received the request is re-sent with the authentication credentials. If authentication succeeds, `update_authenticated` is called to set `is_authenticated` `True` for the URI, so that subsequent requests to the URI or any of its super-URIs will automatically include the authentication credentials.

Novo na versão 3.5: Added `is_authenticated` support.

class urllib.request.**HTTPBasicAuthHandler**(*password_mgr=None*)
Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported. HTTPBasicAuthHandler will raise a *ValueError* when presented with a wrong Authentication scheme.

class urllib.request.**ProxyBasicAuthHandler**(*password_mgr=None*)
Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib.request.**AbstractDigestAuthHandler**(*password_mgr=None*)
This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib.request.**HTTPDigestAuthHandler**(*password_mgr=None*)
Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported. When both Digest Authentication Handler and Basic Authentication Handler are both added, Digest Authentication is always tried first. If the Digest Authentication returns a 40x response again, it is sent to Basic Authentication handler to Handle. This Handler method will raise a *ValueError* when presented with an authentication scheme other than Digest or Basic.

Alterado na versão 3.3: Raise *ValueError* on unsupported Authentication Scheme.

class urllib.request.**ProxyDigestAuthHandler**(*password_mgr=None*)
Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with *HTTPPasswordMgr*; refer to section *HTTPPasswordMgr Objects* for information on the interface that must be supported.

class urllib.request.**HTTPHandler**
A class to handle opening of HTTP URLs.

class urllib.request.**HTTPSHandler**(*debuglevel=0*, *context=None*, *check_hostname=None*)
A class to handle opening of HTTPS URLs. *context* and *check_hostname* have the same meaning as in *http.client.HTTPSConnection*.

Alterado na versão 3.2: *context* and *check_hostname* were added.

class urllib.request.**FileHandler**
Abre arquivos locais.

class urllib.request.**DataHandler**
Abre dados das URLs.

Novo na versão 3.4.

class urllib.request.**FTPHandler**
Abre URLs de FTP.

class urllib.request.**CacheFTPHandler**
Open FTP URLs, keeping a cache of open FTP connections to minimize delays.

class urllib.request.**UnknownHandler**
A catch-all class to handle unknown URLs.

class urllib.request.**HTTPErrorProcessor**
Process HTTP error responses.

22.6.1 Objeto Request

The following methods describe *Request*'s public interface, and so all may be overridden in subclasses. It also defines several public attributes that can be used by clients to inspect the parsed request.

Request.**full_url**
The original URL passed to the constructor.

Alterado na versão 3.4.

Request.full_url is a property with setter, getter and a deleter. Getting *full_url* returns the original request URL with the fragment, if it was present.

Request.**type**
The URI scheme.

Request.**host**
The URI authority, typically a host, but may also contain a port separated by a colon.

Request.**origin_req_host**
The original host for the request, without port.

Request.**selector**
The URI path. If the *Request* uses a proxy, then selector will be the full URL that is passed to the proxy.

Request.**data**
The entity body for the request, or None if not specified.

Alterado na versão 3.4: Changing value of *Request.data* now deletes "Content-Length" header if it was previously set or calculated.

Request.**unverifiable**
boolean, indicates whether the request is unverifiable as defined by **RFC 2965**.

Request.**method**
The HTTP request method to use. By default its value is *None*, which means that *get_method()* will do its normal computation of the method to be used. Its value can be set (thus overriding the default computation in *get_method()*) either by providing a default value by setting it at the class level in a *Request* subclass, or by passing a value in to the *Request* constructor via the *method* argument.

Novo na versão 3.3.

Alterado na versão 3.4: A default value can now be set in subclasses; previously it could only be set via the constructor argument.

Request.**get_method**()
Return a string indicating the HTTP request method. If *Request.method* is not None, return its value, otherwise return 'GET' if *Request.data* is None, or 'POST' if it's not. This is only meaningful for HTTP requests.

Alterado na versão 3.3: get_method now looks at the value of *Request.method*.

Request.**add_header**(*key*, *val*)
Add another header to the request. Headers are currently ignored by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the *key* collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header.

Request.**add_unredirected_header**(*key*, *header*)
Add a header that will not be added to a redirected request.

Request.**has_header**(*header*)
Return whether the instance has the named header (checks both regular and unredirected).

Request.**remove_header**(*header*)
Remove o cabeçalho nomeado da instância de solicitação (tanto de cabeçalhos regulares como de cabeçalhos não-redirecionados).

Novo na versão 3.4.

Request.**get_full_url**()
Return the URL given in the constructor.

Alterado na versão 3.4.

Returns *Request.full_url*

Request.**set_proxy**(*host*, *type*)
Prepare the request by connecting to a proxy server. The *host* and *type* will replace those of the instance, and the instance's selector will be the original URL given in the constructor.

Request.**get_header**(*header_name*, *default=None*)
Return the value of the given header. If the header is not present, return the default value.

Request.**header_items**()
Return a list of tuples (header_name, header_value) of the Request headers.

Alterado na versão 3.4: The request methods add_data, has_data, get_data, get_type, get_host, get_selector, get_origin_req_host and is_unverifiable that were deprecated since 3.3 have been removed.

22.6.2 OpenerDirector Objects

OpenerDirector instances have the following methods:

OpenerDirector.**add_handler**(*handler*)
handler should be an instance of *BaseHandler*. The following methods are searched, and added to the possible chains (note that HTTP errors are a special case). Note that, in the following, *protocol* should be replaced with the actual protocol to handle, for example `http_response()` would be the HTTP protocol response handler. Also *type* should be replaced with the actual HTTP code, for example `http_error_404()` would handle HTTP 404 errors.

- `<protocol>_open()` — signal that the handler knows how to open *protocol* URLs.

  See *BaseHandler.<protocol>_open()* for more information.

- `http_error_<type>()` — signal that the handler knows how to handle HTTP errors with HTTP error code *type*.

  See *BaseHandler.http_error_<nnn>()* for more information.

- `<protocol>_error()` — signal that the handler knows how to handle errors from (non-`http`) *protocol*.

- `<protocol>_request()` — signal that the handler knows how to pre-process *protocol* requests.

  See *BaseHandler.<protocol>_request()* for more information.

- `<protocol>_response()` — signal that the handler knows how to post-process *protocol* responses.

  See *BaseHandler.<protocol>_response()* for more information.

OpenerDirector.**open**(*url*, *data=None*[, *timeout*])
Open the given *url* (which can be a request object or a string), optionally passing the given *data*. Arguments, return values and exceptions raised are the same as those of *urlopen()* (which simply calls the *open()* method on the currently installed global *OpenerDirector*). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). The timeout feature actually works only for HTTP, HTTPS and FTP connections).

OpenerDirector.**error**(*proto*, **args*)
Handle an error of the given protocol. This will call the registered error handlers for the given protocol with the given arguments (which are protocol specific). The HTTP protocol is a special case which uses the HTTP response code to determine the specific error handler; refer to the `http_error_<type>()` methods of the handler classes.

Return values and exceptions raised are the same as those of *urlopen()*.

OpenerDirector objects open URLs in three stages:

The order in which these methods are called within each stage is determined by sorting the handler instances.

1. Every handler with a method named like `<protocol>_request()` has that method called to pre-process the request.

2. Handlers with a method named like `<protocol>_open()` are called to handle the request. This stage ends when a handler either returns a non-*None* value (ie. a response), or raises an exception (usually *URLError*). Exceptions are allowed to propagate.

   In fact, the above algorithm is first tried for methods named `default_open()`. If all such methods return *None*, the algorithm is repeated for methods named like `<protocol>_open()`. If all such methods return *None*, the algorithm is repeated for methods named `unknown_open()`.

   Note that the implementation of these methods may involve calls of the parent *OpenerDirector* instance's *open()* and *error()* methods.

3. Every handler with a method named like `<protocol>_response()` has that method called to post-process the response.

22.6.3 BaseHandler Objects

BaseHandler objects provide a couple of methods that are directly useful, and others that are meant to be used by derived classes. These are intended for direct use:

`BaseHandler.`**`add_parent`**`(`*director*`)`
Add a director as parent.

`BaseHandler.`**`close`**`()`
Remove any parents.

The following attribute and methods should only be used by classes derived from *BaseHandler*.

Nota: The convention has been adopted that subclasses defining `<protocol>_request()` or `<protocol>_response()` methods are named `*Processor`; all others are named `*Handler`.

`BaseHandler.`**`parent`**
A valid *OpenerDirector*, which can be used to open using a different protocol, or handle errors.

`BaseHandler.`**`default_open`**`(`*req*`)`
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to catch all URLs.

This method, if implemented, will be called by the parent *OpenerDirector*. It should return a file-like object as described in the return value of the *open()* of *OpenerDirector*, or `None`. It should raise *URLError*, unless a truly exceptional thing happens (for example, *MemoryError* should not be mapped to `URLError`).

This method will be called before any protocol-specific open method.

`BaseHandler.<protocol>_open(req)`
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to handle URLs with the given protocol.

This method, if defined, will be called by the parent *OpenerDirector*. Return values should be the same as for `default_open()`.

`BaseHandler.`**`unknown_open`**`(`*req*`)`
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to catch all URLs with no specific registered handler to open it.

This method, if implemented, will be called by the *parent OpenerDirector*. Return values should be the same as for *default_open()*.

`BaseHandler.`**`http_error_default`**`(`*req*, *fp*, *code*, *msg*, *hdrs*`)`
This method is *not* defined in *BaseHandler*, but subclasses should override it if they intend to provide a catch-all for otherwise unhandled HTTP errors. It will be called automatically by the *OpenerDirector* getting the error, and should not normally be called in other circumstances.

req will be a *Request* object, *fp* will be a file-like object with the HTTP error body, *code* will be the three-digit code of the error, *msg* will be the user-visible explanation of the code and *hdrs* will be a mapping object with the headers of the error.

Return values and exceptions raised should be the same as those of *urlopen()*.

`BaseHandler.http_error_<nnn>(req, fp, code, msg, hdrs)`
nnn should be a three-digit HTTP error code. This method is also not defined in *BaseHandler*, but will be called, if it exists, on an instance of a subclass, when an HTTP error with code *nnn* occurs.

Subclasses should override this method to handle specific HTTP errors.

Arguments, return values and exceptions raised should be the same as for `http_error_default()`.

BaseHandler.<protocol>_request(req)
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to pre-process requests of the given protocol.

This method, if defined, will be called by the parent *OpenerDirector*. *req* will be a *Request* object. The return value should be a *Request* object.

BaseHandler.<protocol>_response(req, response)
This method is *not* defined in *BaseHandler*, but subclasses should define it if they want to post-process responses of the given protocol.

This method, if defined, will be called by the parent *OpenerDirector*. *req* will be a *Request* object. *response* will be an object implementing the same interface as the return value of *urlopen()*. The return value should implement the same interface as the return value of *urlopen()*.

22.6.4 HTTPRedirectHandler Objects

Nota: Some HTTP redirections require action from this module's client code. If this is the case, *HTTPError* is raised. See **RFC 2616** for details of the precise meanings of the various redirection codes.

An `HTTPError` exception raised as a security consideration if the HTTPRedirectHandler is presented with a redirected URL which is not an HTTP, HTTPS or FTP URL.

`HTTPRedirectHandler.`**redirect_request**(*req*, *fp*, *code*, *msg*, *hdrs*, *newurl*)
Return a *Request* or `None` in response to a redirect. This is called by the default implementations of the `http_error_30*()` methods when a redirection is received from the server. If a redirection should take place, return a new *Request* to allow `http_error_30*()` to perform the redirect to *newurl*. Otherwise, raise *HTTPError* if no other handler should try to handle this URL, or return `None` if you can't but another handler might.

Nota: The default implementation of this method does not strictly follow **RFC 2616**, which says that 301 and 302 responses to `POST` requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a `GET`, and the default implementation reproduces this behavior.

`HTTPRedirectHandler.`**http_error_301**(*req*, *fp*, *code*, *msg*, *hdrs*)
Redirect to the `Location:` or `URI:` URL. This method is called by the parent *OpenerDirector* when getting an HTTP 'moved permanently' response.

`HTTPRedirectHandler.`**http_error_302**(*req*, *fp*, *code*, *msg*, *hdrs*)
The same as *http_error_301()*, but called for the 'found' response.

`HTTPRedirectHandler.`**http_error_303**(*req*, *fp*, *code*, *msg*, *hdrs*)
The same as *http_error_301()*, but called for the 'see other' response.

`HTTPRedirectHandler.`**http_error_307**(*req*, *fp*, *code*, *msg*, *hdrs*)
The same as *http_error_301()*, but called for the 'temporary redirect' response.

22.6.5 HTTPCookieProcessor Objects

HTTPCookieProcessor instances have one attribute:

HTTPCookieProcessor.**cookiejar**
The *http.cookiejar.CookieJar* in which cookies are stored.

22.6.6 ProxyHandler Objects

ProxyHandler.<protocol>_open(request)
The *ProxyHandler* will have a method `<protocol>_open()` for every *protocol* which has a proxy in the *proxies* dictionary given in the constructor. The method will modify requests to go through the proxy, by calling `request.set_proxy()`, and call the next handler in the chain to actually execute the protocol.

22.6.7 HTTPPasswordMgr Objects

These methods are available on *HTTPPasswordMgr* and *HTTPPasswordMgrWithDefaultRealm* objects.

HTTPPasswordMgr.**add_password**(*realm*, *uri*, *user*, *passwd*)
uri can be either a single URI, or a sequence of URIs. *realm*, *user* and *passwd* must be strings. This causes `(user, passwd)` to be used as authentication tokens when authentication for *realm* and a super-URI of any of the given URIs is given.

HTTPPasswordMgr.**find_user_password**(*realm*, *authuri*)
Get user/password for given realm and URI, if any. This method will return `(None, None)` if there is no matching user/password.

For *HTTPPasswordMgrWithDefaultRealm* objects, the realm `None` will be searched if the given *realm* has no matching user/password.

22.6.8 HTTPPasswordMgrWithPriorAuth Objects

This password manager extends *HTTPPasswordMgrWithDefaultRealm* to support tracking URIs for which authentication credentials should always be sent.

HTTPPasswordMgrWithPriorAuth.**add_password**(*realm*, *uri*, *user*, *passwd*, *is_authenticated=False*)
realm, *uri*, *user*, *passwd* are as for *HTTPPasswordMgr.add_password()*. *is_authenticated* sets the initial value of the `is_authenticated` flag for the given URI or list of URIs. If *is_authenticated* is specified as `True`, *realm* is ignored.

HTTPPasswordMgr.**find_user_password**(*realm*, *authuri*)
Same as for *HTTPPasswordMgrWithDefaultRealm* objects

HTTPPasswordMgrWithPriorAuth.**update_authenticated**(*self*, *uri*, *is_authenticated=False*)
Update the `is_authenticated` flag for the given *uri* or list of URIs.

HTTPPasswordMgrWithPriorAuth.**is_authenticated**(*self*, *authuri*)
Returns the current state of the `is_authenticated` flag for the given URI.

22.6.9 AbstractBasicAuthHandler Objects

`AbstractBasicAuthHandler.`**`http_error_auth_reqed`**(*authreq*, *host*, *req*, *headers*)
Handle an authentication request by getting a user/password pair, and re-trying the request. *authreq* should be the name of the header where the information about the realm is included in the request, *host* specifies the URL and path to authenticate for, *req* should be the (failed) `Request` object, and *headers* should be the error headers.

host is either an authority (e.g. `"python.org"`) or a URL containing an authority component (e.g. `"http://python.org/"`). In either case, the authority must not contain a userinfo component (so, `"python.org"` and `"python.org:80"` are fine, `"joe:password@python.org"` is not).

22.6.10 HTTPBasicAuthHandler Objects

`HTTPBasicAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)
Retry the request with authentication information, if available.

22.6.11 ProxyBasicAuthHandler Objects

`ProxyBasicAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)
Retry the request with authentication information, if available.

22.6.12 AbstractDigestAuthHandler Objects

`AbstractDigestAuthHandler.`**`http_error_auth_reqed`**(*authreq*, *host*, *req*, *headers*)
authreq should be the name of the header where the information about the realm is included in the request, *host* should be the host to authenticate to, *req* should be the (failed) `Request` object, and *headers* should be the error headers.

22.6.13 HTTPDigestAuthHandler Objects

`HTTPDigestAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)
Retry the request with authentication information, if available.

22.6.14 ProxyDigestAuthHandler Objects

`ProxyDigestAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)
Retry the request with authentication information, if available.

22.6.15 HTTPHandler Objects

`HTTPHandler.`**`http_open`**(*req*)
Send an HTTP request, which can be either GET or POST, depending on `req.has_data()`.

22.6.16 Objetos HTTPSHandler

`HTTPSHandler.`**`https_open`**`(req)`
Send an HTTPS request, which can be either GET or POST, depending on `req.has_data()`.

22.6.17 FileHandler Objects

`FileHandler.`**`file_open`**`(req)`
Open the file locally, if there is no host name, or the host name is `'localhost'`.

Alterado na versão 3.2: This method is applicable only for local hostnames. When a remote hostname is given, an *URLError* is raised.

22.6.18 DataHandler Objects

`DataHandler.`**`data_open`**`(req)`
Read a data URL. This kind of URL contains the content encoded in the URL itself. The data URL syntax is specified in **RFC 2397**. This implementation ignores white spaces in base64 encoded data URLs so the URL may be wrapped in whatever source file it comes from. But even though some browsers don't mind about a missing padding at the end of a base64 encoded data URL, this implementation will raise an *ValueError* in that case.

22.6.19 FTPHandler Objects

`FTPHandler.`**`ftp_open`**`(req)`
Open the FTP file indicated by *req*. The login is always done with empty username and password.

22.6.20 CacheFTPHandler Objects

CacheFTPHandler objects are *FTPHandler* objects with the following additional methods:

`CacheFTPHandler.`**`setTimeout`**`(t)`
Set timeout of connections to *t* seconds.

`CacheFTPHandler.`**`setMaxConns`**`(m)`
Set maximum number of cached connections to *m*.

22.6.21 Objetos UnknownHandler

`UnknownHandler.`**`unknown_open`**`()`
Raise a *URLError* exception.

22.6.22 HTTPErrorProcessor Objects

`HTTPErrorProcessor.`**`http_response`**(*request*, *response*)
Process HTTP error responses.

For 200 error codes, the response object is returned immediately.

For non-200 error codes, this simply passes the job on to the `http_error_<type>()` handler methods, via *`OpenerDirector.error()`*. Eventually, *`HTTPDefaultErrorHandler`* will raise an *`HTTPError`* if no other handler handles the error.

`HTTPErrorProcessor.`**`https_response`**(*request*, *response*)
Process HTTPS error responses.

The behavior is same as *`http_response()`*.

22.6.23 Exemplos

In addition to the examples below, more examples are given in urllib-howto.

This example gets the python.org main page and displays the first 300 bytes of it.

```
>>> import urllib.request
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(300))
...
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n\n<html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">\n\n<head>\n
<meta http-equiv="content-type" content="text/html; charset=utf-8" />\n
<title>Python Programming '
```

Note that urlopen returns a bytes object. This is because there is no way for urlopen to automatically determine the encoding of the byte stream it receives from the HTTP server. In general, a program will decode the returned bytes object to string once it determines or guesses the appropriate encoding.

The following W3C document, https://www.w3.org/International/O-charset, lists the various ways in which an (X)HTML or an XML document could have specified its encoding information.

As the python.org website uses *utf-8* encoding as specified in its meta tag, we will use the same for decoding the bytes object.

```
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(100).decode('utf-8'))
...
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

It is also possible to achieve the same result without using the *context manager* approach.

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')
>>> print(f.read(100).decode('utf-8'))
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

In the following example, we are sending a data-stream to the stdin of a CGI and reading the data it returns to us. Note that this example will only work when the Python installation supports SSL.

```
>>> import urllib.request
>>> req = urllib.request.Request(url='https://localhost/cgi-bin/test.cgi',
...                       data=b'This data is passed to stdin of the CGI')
>>> with urllib.request.urlopen(req) as f:
...     print(f.read().decode('utf-8'))
...
Got Data: "This data is passed to stdin of the CGI"
```

The code for the sample CGI used in the above example is:

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain\n\nGot Data: "%s"' % data)
```

Here is an example of doing a `PUT` request using *Request*:

```
import urllib.request
DATA = b'some data'
req = urllib.request.Request(url='http://localhost:8080', data=DATA,method='PUT')
with urllib.request.urlopen(req) as f:
    pass
print(f.status)
print(f.reason)
```

Use of Basic HTTP Authentication:

```
import urllib.request
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib.request.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib.request.install_opener(opener)
urllib.request.urlopen('http://www.example.com/login.html')
```

build_opener() provides many handlers by default, including a *ProxyHandler*. By default, *ProxyHandler* uses the environment variables named <scheme>_proxy, where <scheme> is the URL scheme involved. For example, the `http_proxy` environment variable is read to obtain the HTTP proxy's URL.

This example replaces the default *ProxyHandler* with one that uses programmatically-supplied proxy URLs, and adds proxy authorization support with *ProxyBasicAuthHandler*.

```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
# This time, rather than install the OpenerDirector, we use it directly:
opener.open('http://www.example.com/login.html')
```

Adding HTTP headers:

Use the *headers* argument to the *Request* constructor, or:

```
import urllib.request
req = urllib.request.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
# Customize the default User-Agent header value:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . . .)')
r = urllib.request.urlopen(req)
```

OpenerDirector automatically adds a *User-Agent* header to every *Request*. To change this:

```
import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

Also, remember that a few standard headers (*Content-Length*, *Content-Type* and *Host*) are added when the *Request* is passed to *urlopen()* (or *OpenerDirector.open()*).

Here is an example session that uses the `GET` method to retrieve a URL containing parameters:

```
>>> import urllib.request
>>> import urllib.parse
>>> params = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> url = "http://www.musi-cal.com/cgi-bin/query?%s" % params
>>> with urllib.request.urlopen(url) as f:
...     print(f.read().decode('utf-8'))
...
```

The following example uses the `POST` method instead. Note that params output from urlencode is encoded to bytes before it is sent to urlopen as data:

```
>>> import urllib.request
>>> import urllib.parse
>>> data = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> data = data.encode('ascii')
>>> with urllib.request.urlopen("http://requestb.in/xrbl82xr", data) as f:
...     print(f.read().decode('utf-8'))
...
```

The following example uses an explicitly specified HTTP proxy, overriding environment settings:

```
>>> import urllib.request
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.request.FancyURLopener(proxies)
>>> with opener.open("http://www.python.org") as f:
...     f.read().decode('utf-8')
...
```

The following example uses no proxies at all, overriding environment settings:

```
>>> import urllib.request
>>> opener = urllib.request.FancyURLopener({})
>>> with opener.open("http://www.python.org/") as f:
...     f.read().decode('utf-8')
...
```

22.6.24 Legacy interface

The following functions and classes are ported from the Python 2 module `urllib` (as opposed to `urllib2`). They might become deprecated at some point in the future.

`urllib.request.`**`urlretrieve`**(*url*, *filename=None*, *reporthook=None*, *data=None*)

Copy a network object denoted by a URL to a local file. If the URL points to a local file, the object will not be copied unless filename is supplied. Return a tuple `(filename, headers)` where *filename* is the local file name under which the object can be found, and *headers* is whatever the `info()` method of the object returned by *urlopen()* returned (for a remote object). Exceptions are the same as for *urlopen()*.

The second argument, if present, specifies the file location to copy to (if absent, the location will be a tempfile with a generated name). The third argument, if present, is a callable that will be called once on establishment of the network connection and once after each block read thereafter. The callable will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be `-1` on older FTP servers which do not return a file size in response to a retrieval request.

The following example illustrates the most common usage scenario:

```
>>> import urllib.request
>>> local_filename, headers = urllib.request.urlretrieve('http://python.org/')
>>> html = open(local_filename)
>>> html.close()
```

If the *url* uses the `http:` scheme identifier, the optional *data* argument may be given to specify a `POST` request (normally the request type is `GET`). The *data* argument must be a bytes object in standard *application/x-www-form-urlencoded* format; see the *urllib.parse.urlencode()* function.

urlretrieve() will raise `ContentTooShortError` when it detects that the amount of data available was less than the expected amount (which is the size reported by a *Content-Length* header). This can occur, for example, when the download is interrupted.

The *Content-Length* is treated as a lower bound: if there's more data to read, urlretrieve reads more data, but if less data is available, it raises the exception.

You can still retrieve the downloaded data in this case, it is stored in the `content` attribute of the exception instance.

If no *Content-Length* header was supplied, urlretrieve can not check the size of the data it has downloaded, and just returns it. In this case you just have to assume that the download was successful.

`urllib.request.`**`urlcleanup`**()

Cleans up temporary files that may have been left behind by previous calls to *urlretrieve()*.

`class` `urllib.request.`**`URLopener`**(*proxies=None*, ***x509*)

Obsoleto desde a versão 3.3.

Base class for opening and reading URLs. Unless you need to support opening objects using schemes other than `http:`, `ftp:`, or `file:`, you probably want to use *FancyURLopener*.

By default, the *URLopener* class sends a *User-Agent* header of `urllib/VVV`, where *VVV* is the *urllib* version number. Applications can define their own *User-Agent* header by subclassing *URLopener* or *FancyURLopener* and setting the class attribute *version* to an appropriate string value in the subclass definition.

The optional *proxies* parameter should be a dictionary mapping scheme names to proxy URLs, where an empty dictionary turns proxies off completely. Its default value is `None`, in which case environmental proxy settings will be used if present, as discussed in the definition of *urlopen()*, above.

Additional keyword parameters, collected in *x509*, may be used for authentication of the client when using the `https:` scheme. The keywords *key_file* and *cert_file* are supported to provide an SSL key and certificate; both are needed to support client authentication.

URLopener objects will raise an *OSError* exception if the server returns an error code.

open(*fullurl*, *data=None*)
Open *fullurl* using the appropriate protocol. This method sets up cache and proxy information, then calls the appropriate open method with its input arguments. If the scheme is not recognized, *open_unknown()* is called. The *data* argument has the same meaning as the *data* argument of *urlopen()*.

This method always quotes *fullurl* using *quote()*.

open_unknown(*fullurl*, *data=None*)
Overridable interface to open unknown URL types.

retrieve(*url*, *filename=None*, *reporthook=None*, *data=None*)
Retrieves the contents of *url* and places it in *filename*. The return value is a tuple consisting of a local filename and either an *email.message.Message* object containing the response headers (for remote URLs) or `None` (for local URLs). The caller must then open and read the contents of *filename*. If *filename* is not given and the URL refers to a local file, the input filename is returned. If the URL is non-local and *filename* is not given, the filename is the output of *tempfile.mktemp()* with a suffix that matches the suffix of the last path component of the input URL. If *reporthook* is given, it must be a function accepting three numeric parameters: A chunk number, the maximum size chunks are read in and the total size of the download (-1 if unknown). It will be called once at the start and after each chunk of data is read from the network. *reporthook* is ignored for local URLs.

If the *url* uses the `http:` scheme identifier, the optional *data* argument may be given to specify a `POST` request (normally the request type is `GET`). The *data* argument must in standard *application/x-www-form-urlencoded* format; see the *urllib.parse.urlencode()* function.

version
Variable that specifies the user agent of the opener object. To get *urllib* to tell servers that it is a particular user agent, set this in a subclass as a class variable or in the constructor before calling the base constructor.

class urllib.request.**FancyURLopener**(...)
Obsoleto desde a versão 3.3.

FancyURLopener subclasses *URLopener* providing default handling for the following HTTP response codes: 301, 302, 303, 307 and 401. For the 30x response codes listed above, the *Location* header is used to fetch the actual URL. For 401 response codes (authentication required), basic HTTP authentication is performed. For the 30x response codes, recursion is bounded by the value of the *maxtries* attribute, which defaults to 10.

For all other response codes, the method `http_error_default()` is called which you can override in subclasses to handle the error appropriately.

Nota: According to the letter of **RFC 2616**, 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and *urllib* reproduces this behaviour.

The parameters to the constructor are the same as those for *URLopener*.

Nota: When performing basic authentication, a *FancyURLopener* instance calls its *prompt_user_passwd()* method. The default implementation asks the users for the required information on the controlling terminal. A subclass may override this method to support more appropriate behavior if needed.

The *FancyURLopener* class offers one additional method that should be overloaded to provide the appropriate behavior:

prompt_user_passwd(*host*, *realm*)
Return information needed to authenticate the user at the given host in the specified security realm. The return value should be a tuple, `(user, password)`, which can be used for basic authentication.

The implementation prompts for this information on the terminal; an application should override this method to use an appropriate interaction model in the local environment.

22.6.25 urllib.request Restrictions

- Currently, only the following protocols are supported: HTTP (versions 0.9 and 1.0), FTP, local files, and data URLs.

  Alterado na versão 3.4: Added support for data URLs.

- The caching feature of *urlretrieve()* has been disabled until someone finds the time to hack proper processing of Expiration time headers.
- There should be a function to query whether a particular URL is in the cache.
- For backward compatibility, if a URL appears to point to a local file but the file can't be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.
- The *urlopen()* and *urlretrieve()* functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive Web client using these functions without using threads.
- The data returned by *urlopen()* or *urlretrieve()* is the raw data returned by the server. This may be binary data (such as an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the *Content-Type* header. If the returned data is HTML, you can use the module *html.parser* to parse it.
- The code handling the FTP protocol cannot differentiate between a file and a directory. This can lead to unexpected behavior when attempting to read a URL that points to a file that is not accessible. If the URL ends in a `/`, it is assumed to refer to a directory and will be handled accordingly. But if an attempt to read a file leads to a 550 error (meaning the URL cannot be found or is not accessible, often for permission reasons), then the path is treated as a directory in order to handle the case when a directory is specified by a URL but the trailing `/` has been left off. This can cause misleading results when you try to fetch a file whose read permissions make it inaccessible; the FTP code will try to read it, fail with a 550 error, and then perform a directory listing for the unreadable file. If fine-grained control is needed, consider using the *ftplib* module, subclassing *FancyURLopener*, or changing *_urlopener* to meet your needs.

22.7 urllib.response — Response classes used by urllib

The *urllib.response* module defines functions and classes which define a minimal file like interface, including `read()` and `readline()`. The typical response object is an addinfourl instance, which defines an `info()` method and that returns headers and a `geturl()` method that returns the url. Functions defined by this module are used internally by the *urllib.request* module.

22.8 `urllib.parse` — Analisa URLs para componentes

Código Fonte: Lib/urllib/parse.py

Este módulo define uma interface padrão para quebrar strings de Uniform Resource Locator (URL) em componentes (esquema de endereçamento, local de rede, caminho etc.), para combinar os componentes de volta em uma string de URL e para converter uma "URL relativo" em uma URL absoluta dado uma "URL base".

O módulo foi projetado para corresponder ao RFC da Internet sobre Localizadores de Recursos Uniformes Relativos, ou URL relativa. Ele tem suporte aos seguintes esquemas de URL: `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `mailto`, `mms`, `news`, `nntp`, `prospero`, `rsync`, `rtsp`, `rtspu`, `sftp`, `shttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`, `ws`, `wss`.

O módulo *`urllib.parse`* define funções que se enquadram em duas grandes categorias: análise de URL e colocação de aspas na URL. Eles são abordados em detalhes nas seções a seguir.

22.8.1 Análise de URL

As funções de análise de URL se concentram na divisão de uma string de URL em seus componentes ou na combinação de componentes de URL em uma string de URL.

`urllib.parse.`**`urlparse`**(*urlstring*, *scheme=''*, *allow_fragments=True*)

Parse a URL into six components, returning a 6-item *named tuple*. This corresponds to the general structure of a URL: `scheme://netloc/path;parameters?query#fragment`. Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (for example, the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the *path* component, which is retained if present. For example:

```
>>> from urllib.parse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> o.scheme
'http'
>>> o.port
80
>>> o.geturl()
'http://www.cwi.nl:80/%7Eguido/Python.html'
```

Seguindo as especificações de sintaxe em **RFC 1808**, o urlparse reconhece um netloc apenas se for introduzido apropriadamente por '//'. Caso contrário, presume-se que a entrada seja uma URL relativa e, portanto, comece com um componente de caminho.

```
>>> from urllib.parse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='',
            query='', fragment='')
```

O argumento *scheme* fornece o esquema de endereçamento padrão, a ser usado apenas se o URL não especificar um. Deve ser do mesmo tipo (texto ou bytes) que *urlstring*, exceto que o valor padrão `''` é sempre permitido e é automaticamente convertido para `b''` se apropriado.

Se o argumento *allow_fragments* for falso, os identificadores de fragmento não serão reconhecidos. Em vez disso, eles são analisados como parte do caminho, parâmetros ou componente de consulta, e `fragment` é definido como a string vazia no valor de retorno.

O valor de retorno é uma tupla nomeada, o que significa que seus itens podem ser acessados por índice ou como atributos nomeados, que são:

Atributo	Index	Valor	Valor, se não presente
`scheme`	0	Especificador do esquema da URL	parâmetro *scheme*
`netloc`	1	Parte da localização na rede	string vazia
`path`	2	Caminho hierárquico	string vazia
`params`	3	Parâmetros para o último elemento de caminho	string vazia
`query`	4	Componente da consulta	string vazia
`fragment`	5	Identificador do fragmento	string vazia
`username`		Nome do usuário	*None*
`password`		Senha	*None*
`hostname`		Nome de máquina (em minúsculo)	*None*
`port`		Número da porta como inteiro, se presente	*None*

Ler o atributo `port` irá levantar uma *ValueError* se uma porta inválida for especificada no URL. Veja a seção *Structured Parse Results* para mais informações sobre o objeto de resultado.

Colchetes sem correspondência no atributo `netloc` levantará uma *ValueError*.

Caracteres no atributo `netloc` que se decompõem sob a normalização NFKC (como usado pela codificação IDNA) em qualquer um dos `/`, `?`, `#`, `@` ou `:` vai levantar uma *ValueError*. Se a URL for decomposta antes da análise, nenhum erro será levantado.

Como é o caso com todas as tuplas nomeadas, a subclasse tem alguns métodos e atributos adicionais que são particularmente úteis. Um desses métodos é `_replace()`. O método `_replace()` retornará um novo objeto ParseResult substituindo os campos especificados por novos valores.

```
>>> from urllib.parse import urlparse
>>> u = urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
>>> u
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> u._replace(scheme='http')
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
```

Aviso: *urlparse()* does not perform validation. See *URL parsing security* for details.

Alterado na versão 3.2: Adicionados recursos de análise de URL IPv6.

Alterado na versão 3.3: O fragmento agora é analisado para todos os esquemas de URL (a menos que *allow_fragment* seja falso), de acordo com a **RFC 3986**. Anteriormente, existia uma lista branca de esquemas que suportam fragmentos.

Alterado na versão 3.6: Números de porta fora do intervalo agora levantam *ValueError*, em vez de retornar *None*.

Alterado na versão 3.7.3: Os caracteres que afetam a análise de netloc sob normalização NFKC agora levantarão `ValueError`.

urllib.parse.**parse_qs**(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

Analisa uma string de consulta fornecida como um argumento de string (dados do tipo `application/x-www-form-urlencoded`). Os dados são retornados como um dicionário. As chaves de dicionário são os nomes de variáveis de consulta exclusivos e os valores são listas de valores para cada nome.

O argumento opcional *keep_blank_values* é um sinalizador que indica se os valores em branco em consultas codificadas por porcentagem devem ser tratados como strings em branco. Um valor verdadeiro indica que os espaços em branco devem ser mantidos como strings em branco. O valor falso padrão indica que os valores em branco devem ser ignorados e tratados como se não tivessem sido incluídos.

O argumento opcional *strict_parsing* é um sinalizador que indica o que fazer com os erros de análise. Se falsO (o padrão), os erros são ignorados silenciosamente. Se verdadeiro, os erros levantam uma exceção `ValueError`.

Os parâmetros opcionais *encoding* e *errors* especificam como decodificar sequências codificadas em porcentagem em caracteres Unicode, conforme aceito pelo método `bytes.decode()`.

O argumento opcional *max_num_fields* é o número máximo de campos a serem lidos. Se definido, então levanta um `ValueError` se houver mais de *max_num_fields* campos lidos.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

Use a função `urllib.parse.urlencode()` (com o parâmetro `doseq` definido como `True`) para converter esses dicionários em strings de consulta.

Alterado na versão 3.2: Adicionado os parâmetros *encoding* e *errors*.

Alterado na versão 3.7.2: Adicionado o parâmetro *max_num_fields*.

Alterado na versão 3.7.10: Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.7.10 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

urllib.parse.**parse_qsl**(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

Parse a query string given as a string argument (data of type `application/x-www-form-urlencoded`). Data are returned as a list of name, value pairs.

O argumento opcional *keep_blank_values* é um sinalizador que indica se os valores em branco em consultas codificadas por porcentagem devem ser tratados como strings em branco. Um valor verdadeiro indica que os espaços em branco devem ser mantidos como strings em branco. O valor falso padrão indica que os valores em branco devem ser ignorados e tratados como se não tivessem sido incluídos.

O argumento opcional *strict_parsing* é um sinalizador que indica o que fazer com os erros de análise. Se falsO (o padrão), os erros são ignorados silenciosamente. Se verdadeiro, os erros levantam uma exceção `ValueError`.

Os parâmetros opcionais *encoding* e *errors* especificam como decodificar sequências codificadas em porcentagem em caracteres Unicode, conforme aceito pelo método `bytes.decode()`.

O argumento opcional *max_num_fields* é o número máximo de campos a serem lidos. Se definido, então levanta um `ValueError` se houver mais de *max_num_fields* campos lidos.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

Use the `urllib.parse.urlencode()` function to convert such lists of pairs into query strings.

Alterado na versão 3.2: Adicionado os parâmetros *encoding* e *errors*.

Alterado na versão 3.7.2: Adicionado o parâmetro *max_num_fields*.

Alterado na versão 3.7.10: Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.7.10 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

`urllib.parse.`**`urlunparse`**(*parts*)
Construct a URL from a tuple as returned by `urlparse()`. The *parts* argument can be any six-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a `?` with an empty query; the RFC states that these are equivalent).

`urllib.parse.`**`urlsplit`**(*urlstring*, *scheme=''*, *allow_fragments=True*)
This is similar to *`urlparse()`*, but does not split the params from the URL. This should generally be used instead of *`urlparse()`* if the more recent URL syntax allowing parameters to be applied to each segment of the *path* portion of the URL (see **RFC 2396**) is wanted. A separate function is needed to separate the path segments and parameters. This function returns a 5-item *named tuple*:

```
(addressing scheme, network location, path, query, fragment identifier).
```

The return value is a *named tuple*, its items can be accessed by index or as named attributes:

Atributo	Index	Valor	Valor, se não presente
`scheme`	0	Especificador do esquema da URL	parâmetro *scheme*
`netloc`	1	Parte da localização na rede	string vazia
`path`	2	Caminho hierárquico	string vazia
`query`	3	Componente da consulta	string vazia
`fragment`	4	Identificador do fragmento	string vazia
`username`		Nome do usuário	*`None`*
`password`		Senha	*`None`*
`hostname`		Nome de máquina (em minúsculo)	*`None`*
`port`		Número da porta como inteiro, se presente	*`None`*

Ler o atributo `port` irá levantar uma *`ValueError`* se uma porta inválida for especificada no URL. Veja a seção *Structured Parse Results* para mais informações sobre o objeto de resultado.

Colchetes sem correspondência no atributo `netloc` levantará uma *`ValueError`*.

Caracteres no atributo `netloc` que se decompõem sob a normalização NFKC (como usado pela codificação IDNA) em qualquer um dos `/`, `?`, `#`, `@` ou `:` vai levantar uma *`ValueError`*. Se a URL for decomposta antes da análise, nenhum erro será levantado.

Following some of the WHATWG spec that updates RFC 3986, leading C0 control and space characters are stripped from the URL. `\n`, `\r` and tab `\t` characters are removed from the URL at any position.

Aviso: *`urlsplit()`* does not perform validation. See *URL parsing security* for details.

Alterado na versão 3.6: Números de porta fora do intervalo agora levantam *`ValueError`*, em vez de retornar *`None`*.

Alterado na versão 3.7.3: Os caracteres que afetam a análise de netloc sob normalização NFKC agora levantarão *`ValueError`*.

Alterado na versão 3.7.11: ASCII newline and tab characters are stripped from the URL.

Alterado na versão 3.7.17: Leading WHATWG C0 control and space characters are stripped from the URL.

`urllib.parse.`**`urlunsplit`**(*parts*)
Combine the elements of a tuple as returned by *`urlsplit()`* into a complete URL as a string. The *parts* argument can be any five-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was

parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).

urllib.parse.**urljoin**(*base*, *url*, *allow_fragments=True*)
Construct a full ("absolute") URL by combining a "base URL" (*base*) with another URL (*url*). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL. For example:

```
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The *allow_fragments* argument has the same meaning and default as for *urlparse()*.

Nota: If *url* is an absolute URL (that is, starting with `//` or `scheme://`), the *url*'s host name and/or scheme will be present in the result. For example:

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

If you do not want that behavior, preprocess the *url* with *urlsplit()* and *urlunsplit()*, removing possible *scheme* and *netloc* parts.

Alterado na versão 3.5: Behaviour updated to match the semantics defined in **RFC 3986**.

urllib.parse.**urldefrag**(*url*)
If *url* contains a fragment identifier, return a modified version of *url* with no fragment identifier, and the fragment identifier as a separate string. If there is no fragment identifier in *url*, return *url* unmodified and an empty string.

The return value is a *named tuple*, its items can be accessed by index or as named attributes:

Atributo	Index	Valor	Valor, se não presente
`url`	0	URL with no fragment	string vazia
`fragment`	1	Identificador do fragmento	string vazia

See section *Structured Parse Results* for more information on the result object.

Alterado na versão 3.2: Result is a structured object rather than a simple 2-tuple.

22.8.2 URL parsing security

The *urlsplit()* and *urlparse()* APIs do not perform **validation** of inputs. They may not raise errors on inputs that other applications consider invalid. They may also succeed on some inputs that might not be considered URLs elsewhere. Their purpose is for practical functionality rather than purity.

Instead of raising an exception on unusual input, they may instead return some component parts as empty strings. Or components may contain more than perhaps they should.

We recommend that users of these APIs where the values may be used anywhere with security implications code defensively. Do some verification within your code before trusting a returned component part. Does that `scheme` make sense? Is that a sensible `path`? Is there anything strange about that `hostname`? etc.

22.8.3 Analisando bytes codificados em ASCII

The URL parsing functions were originally designed to operate on character strings only. In practice, it is useful to be able to manipulate properly quoted and encoded URLs as sequences of ASCII bytes. Accordingly, the URL parsing functions in this module all operate on *bytes* and *bytearray* objects in addition to *str* objects.

If *str* data is passed in, the result will also contain only *str* data. If *bytes* or *bytearray* data is passed in, the result will contain only *bytes* data.

Attempting to mix *str* data with *bytes* or *bytearray* in a single function call will result in a *TypeError* being raised, while attempting to pass in non-ASCII byte values will trigger *UnicodeDecodeError*.

To support easier conversion of result objects between *str* and *bytes*, all return values from URL parsing functions provide either an `encode()` method (when the result contains *str* data) or a `decode()` method (when the result contains *bytes* data). The signatures of these methods match those of the corresponding *str* and *bytes* methods (except that the default encoding is `'ascii'` rather than `'utf-8'`). Each produces a value of a corresponding type that contains either *bytes* data (for `encode()` methods) or *str* data (for `decode()` methods).

Applications that need to operate on potentially improperly quoted URLs that may contain non-ASCII data will need to do their own decoding from bytes to characters before invoking the URL parsing methods.

The behaviour described in this section applies only to the URL parsing functions. The URL quoting functions use their own rules when producing or consuming byte sequences as detailed in the documentation of the individual URL quoting functions.

Alterado na versão 3.2: URL parsing functions now accept ASCII encoded byte sequences

22.8.4 Structured Parse Results

The result objects from the *urlparse()*, *urlsplit()* and *urldefrag()* functions are subclasses of the *tuple* type. These subclasses add the attributes listed in the documentation for those functions, the encoding and decoding support described in the previous section, as well as an additional method:

`urllib.parse.SplitResult.`**`geturl`**`()`

Return the re-combined version of the original URL as a string. This may differ from the original URL in that the scheme may be normalized to lower case and empty components may be dropped. Specifically, empty parameters, queries, and fragment identifiers will be removed.

For *urldefrag()* results, only empty fragment identifiers will be removed. For *urlsplit()* and *urlparse()* results, all noted changes will be made to the URL returned by this method.

The result of this method remains unchanged if passed back through the original parsing function:

```
>>> from urllib.parse import urlsplit
>>> url = 'HTTP://www.Python.org/doc/#'
>>> r1 = urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
>>> r2 = urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

The following classes provide the implementations of the structured parse results when operating on *str* objects:

`class` `urllib.parse.`**`DefragResult`**`(`*url*, *fragment*`)`

Concrete class for *urldefrag()* results containing *str* data. The `encode()` method returns a *DefragResultBytes* instance.

Novo na versão 3.2.

class urllib.parse.**ParseResult**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)
Concrete class for *urlparse()* results containing *str* data. The `encode()` method returns a *ParseResultBytes* instance.

class urllib.parse.**SplitResult**(*scheme*, *netloc*, *path*, *query*, *fragment*)
Concrete class for *urlsplit()* results containing *str* data. The `encode()` method returns a *SplitResultBytes* instance.

The following classes provide the implementations of the parse results when operating on *bytes* or *bytearray* objects:

class urllib.parse.**DefragResultBytes**(*url*, *fragment*)
Concrete class for *urldefrag()* results containing *bytes* data. The `decode()` method returns a *DefragResult* instance.

Novo na versão 3.2.

class urllib.parse.**ParseResultBytes**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)
Concrete class for *urlparse()* results containing *bytes* data. The `decode()` method returns a *ParseResult* instance.

Novo na versão 3.2.

class urllib.parse.**SplitResultBytes**(*scheme*, *netloc*, *path*, *query*, *fragment*)
Concrete class for *urlsplit()* results containing *bytes* data. The `decode()` method returns a *SplitResult* instance.

Novo na versão 3.2.

22.8.5 URL Quoting

The URL quoting functions focus on taking program data and making it safe for use as URL components by quoting special characters and appropriately encoding non-ASCII text. They also support reversing these operations to recreate the original data from the contents of a URL component if that task isn't already covered by the URL parsing functions above.

urllib.parse.**quote**(*string*, *safe='/'*, *encoding=None*, *errors=None*)
Replace special characters in *string* using the `%xx` escape. Letters, digits, and the characters `'_.-~'` are never quoted. By default, this function is intended for quoting the path section of URL. The optional *safe* parameter specifies additional ASCII characters that should not be quoted — its default value is `'/'`.

string may be either a *str* or a *bytes*.

Alterado na versão 3.7: Moved from **RFC 2396** to **RFC 3986** for quoting URL strings. "~" is now included in the set of unreserved characters.

The optional *encoding* and *errors* parameters specify how to deal with non-ASCII characters, as accepted by the *str.encode()* method. *encoding* defaults to `'utf-8'`. *errors* defaults to `'strict'`, meaning unsupported characters raise a *UnicodeEncodeError*. *encoding* and *errors* must not be supplied if *string* is a *bytes*, or a *TypeError* is raised.

Note that `quote(string, safe, encoding, errors)` is equivalent to `quote_from_bytes(string.encode(encoding, errors), safe)`.

Example: `quote('/El Niño/')` yields `'/El%20Ni%C3%B1o/'`.

urllib.parse.**quote_plus**(*string*, *safe=''*, *encoding=None*, *errors=None*)
Like *quote()*, but also replace spaces by plus signs, as required for quoting HTML form values when building up a query string to go into a URL. Plus signs in the original string are escaped unless they are included in *safe*. It also does not have *safe* default to `'/'`.

Example: `quote_plus('/El Niño/')` yields `'%2FEl+Ni%C3%B1o%2F'`.

`urllib.parse.`**`quote_from_bytes`**(*bytes*, *safe='/'*)
Like *quote()*, but accepts a *bytes* object rather than a *str*, and does not perform string-to-bytes encoding.

Example: `quote_from_bytes(b'a&\xef')` yields `'a%26%EF'`.

`urllib.parse.`**`unquote`**(*string*, *encoding='utf-8'*, *errors='replace'*)
Replace `%xx` escapes by their single-character equivalent. The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the *bytes.decode()* method.

string must be a *str*.

encoding defaults to `'utf-8'`. *errors* defaults to `'replace'`, meaning invalid sequences are replaced by a placeholder character.

Example: `unquote('/El%20Ni%C3%B1o/')` yields `'/El Niño/'`.

`urllib.parse.`**`unquote_plus`**(*string*, *encoding='utf-8'*, *errors='replace'*)
Like *unquote()*, but also replace plus signs by spaces, as required for unquoting HTML form values.

string must be a *str*.

Example: `unquote_plus('/El+Ni%C3%B1o/')` yields `'/El Niño/'`.

`urllib.parse.`**`unquote_to_bytes`**(*string*)
Replace `%xx` escapes by their single-octet equivalent, and return a *bytes* object.

string may be either a *str* or a *bytes*.

If it is a *str*, unescaped non-ASCII characters in *string* are encoded into UTF-8 bytes.

Example: `unquote_to_bytes('a%26%EF')` yields `b'a&\xef'`.

`urllib.parse.`**`urlencode`**(*query*, *doseq=False*, *safe=''*, *encoding=None*, *errors=None*, *quote_via=quote_plus*)
Convert a mapping object or a sequence of two-element tuples, which may contain *str* or *bytes* objects, to a percent-encoded ASCII text string. If the resultant string is to be used as a *data* for POST operation with the *urlopen()* function, then it should be encoded to bytes, otherwise it would result in a *TypeError*.

The resulting string is a series of `key=value` pairs separated by `'&'` characters, where both *key* and *value* are quoted using the *quote_via* function. By default, *quote_plus()* is used to quote the values, which means spaces are quoted as a `'+'` character and '/' characters are encoded as `%2F`, which follows the standard for GET requests (`application/x-www-form-urlencoded`). An alternate function that can be passed as *quote_via* is *quote()*, which will encode spaces as `%20` and not encode '/' characters. For maximum control of what is quoted, use `quote` and specify a value for *safe*.

When a sequence of two-element tuples is used as the *query* argument, the first element of each tuple is a key and the second is a value. The value element in itself can be a sequence and in that case, if the optional parameter *doseq* is evaluates to `True`, individual `key=value` pairs separated by `'&'` are generated for each element of the value sequence for the key. The order of parameters in the encoded string will match the order of parameter tuples in the sequence.

The *safe*, *encoding*, and *errors* parameters are passed down to *quote_via* (the *encoding* and *errors* parameters are only passed when a query element is a *str*).

To reverse this encoding process, *parse_qs()* and *parse_qsl()* are provided in this module to parse query strings into Python data structures.

Refer to *urllib examples* to find out how urlencode method can be used for generating query string for a URL or data for POST.

Alterado na versão 3.2: Query parameter supports bytes and string objects.

Novo na versão 3.5: *quote_via* parameter.

Ver também:

WHATWG - URL Living standard Working Group for the URL Standard that defines URLs, domains, IP addresses, the application/x-www-form-urlencoded format, and their API.

RFC 3986 - Uniform Resource Identifiers This is the current standard (STD66). Any changes to urllib.parse module should conform to this. Certain deviations could be observed, which are mostly for backward compatibility purposes and for certain de-facto parsing requirements as commonly observed in major browsers.

RFC 2732 - Format for Literal IPv6 Addresses in URL's. This specifies the parsing requirements of IPv6 URLs.

RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

RFC 2368 - The mailto URL scheme. Parsing requirements for mailto URL schemes.

RFC 1808 - Relative Uniform Resource Locators This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair number of "Abnormal Examples" which govern the treatment of border cases.

RFC 1738 - Uniform Resource Locators (URL) This specifies the formal syntax and semantics of absolute URLs.

22.9 `urllib.error` — Classes de exceção levantadas por urllib.request

Código Fonte: Lib/urllib/error.py

O módulo *`urllib.error`* define as classes de exceção para exceções levantadas por *`urllib.request`*. A classe de exceção base é *`URLError`*.

As seguintes exceções são levantadas por *`urllib.error`* conforme apropriado:

exception urllib.error.**URLError**
Os manipuladores levantam essa exceção (ou exceções derivadas) quando encontram um problema. É uma subclasse de *`OSError`*.

reason
O motivo desse erro. Pode ser uma string de mensagem ou outra instância de exceção.

Alterado na versão 3.3: *`URLError`* foi tornada uma subclasse de *`OSError`* em vez de *`IOError`*.

exception urllib.error.**HTTPError**
Embora seja uma exceção (uma subclasse de *`URLError`*), uma *`HTTPError`* também pode funcionar como um valor de retorno não excepcional do tipo arquivo (a mesma coisa que *`urlopen()`* retorna). Isso é útil ao lidar com erros de HTTP exóticos, como solicitações de autenticação.

code
Um código de status HTTP conforme definido em **RFC 2616**. Este valor numérico corresponde a um valor encontrado no dicionário de códigos conforme encontrado em *`http.server.BaseHTTPRequestHandler.responses`*.

reason
Geralmente é uma string explicando o motivo desse erro.

headers
Os cabeçalhos de resposta HTTP para a solicitação HTTP que causou a *`HTTPError`*.

Novo na versão 3.4.

exception urllib.error.**ContentTooShortError**(*msg*, *content*)
Esta exceção é levantada quando a função *urlretrieve()* detecta que a quantidade de dados baixados é menor que a quantidade esperada (fornecida pelo cabeçalho *Content-Length*). O atributo `content` armazena os dados baixados (e supostamente truncados).

22.10 urllib.robotparser — Parser for robots.txt

Código Fonte: Lib/urllib/robotparser.py

This module provides a single class, *RobotFileParser*, which answers questions about whether or not a particular user agent can fetch a URL on the Web site that published the `robots.txt` file. For more details on the structure of `robots.txt` files, see http://www.robotstxt.org/orig.html.

class urllib.robotparser.**RobotFileParser**(*url=''*)
This class provides methods to read, parse and answer questions about the `robots.txt` file at *url*.

set_url(*url*)
Sets the URL referring to a `robots.txt` file.

read()
Reads the `robots.txt` URL and feeds it to the parser.

parse(*lines*)
Parses the lines argument.

can_fetch(*useragent*, *url*)
Returns `True` if the *useragent* is allowed to fetch the *url* according to the rules contained in the parsed `robots.txt` file.

mtime()
Returns the time the `robots.txt` file was last fetched. This is useful for long-running web spiders that need to check for new `robots.txt` files periodically.

modified()
Sets the time the `robots.txt` file was last fetched to the current time.

crawl_delay(*useragent*)
Returns the value of the `Crawl-delay` parameter from `robots.txt` for the *useragent* in question. If there is no such parameter or it doesn't apply to the *useragent* specified or the `robots.txt` entry for this parameter has invalid syntax, return `None`.

Novo na versão 3.6.

request_rate(*useragent*)
Returns the contents of the `Request-rate` parameter from `robots.txt` as a *named tuple* `RequestRate(requests, seconds)`. If there is no such parameter or it doesn't apply to the *useragent* specified or the `robots.txt` entry for this parameter has invalid syntax, return `None`.

Novo na versão 3.6.

O exemplo a seguir demonstra o uso básico da classe *RobotFileParser*:

```
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rrate = rp.request_rate("*")
```

(continua na próxima página)

(continuação da página anterior)

```
>>> rrate.requests
3
>>> rrate.seconds
20
>>> rp.crawl_delay("*")
6
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

22.11 `http` — módulos HTTP

Código Fonte: Lib/http/__init__.py

---

http é um pacote que coleta vários módulos para trabalhar com o Protocolo de Transferência de Hipertexto:

- *http.client* é um cliente de protocolo HTTP de baixo-nível; para abertura de URL alto-nível utilize *urllib.request*
- *http.server* contém classes básicas de servidores HTTP baseadas em *socketserver*
- *http.cookies* tem utilidades para implementar gerenciamento de estado com cookies
- *http.cookiejar* provê persistência de cookies

http é também um módulo que define um número de códigos de status HTTP e mensagens associadas através do enum *http.HTTPStatus*:

class http.**HTTPStatus**

Novo na versão 3.5.

Subclasse de *enum.IntEnum* que define um conjunto de códigos de status HTTP, frases de razão e descrições longas escritas em inglês.

Utilização:

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK
<HTTPStatus.OK: 200>
>>> HTTPStatus.OK == 200
True
>>> HTTPStatus.OK.value
200
>>> HTTPStatus.OK.phrase
'OK'
>>> HTTPStatus.OK.description
'Request fulfilled, document follows'
>>> list(HTTPStatus)
[<HTTPStatus.CONTINUE: 100>, <HTTPStatus.SWITCHING_PROTOCOLS: 101>, ...]
```

22.11.1 códigos de status HTTP

Suportados, códigos de status IANA-registered disponíveis em `http.HTTPStatus` são:

Código	Nome da Enumeração	Detalhes
100	CONTINUE	HTTP/1.1 **RFC 7231**, Seção 6.2.1
101	SWITCHING_PROTOCOLS	HTTP/1.1 **RFC 7231**, Seçãp 6.2.2
102	PROCESSING	WebDAV **RFC 2518**, Seção 10.1
200	OK	HTTP/1.1 **RFC 7231**, Seção 6.3.1
201	CREATED	HTTP/1.1 **RFC 7231**, Seção 6.3.2
202	ACCEPTED	HTTP/1.1 **RFC 7231**, Seção 6.3.3
203	NON_AUTHORITATIVE_INFORMATION	HTTP/1.1 **RFC 7231**, Seção 6.3.4
204	NO_CONTENT	HTTP/1.1 **RFC 7231**, Seção 6.3.5
205	RESET_CONTENT	HTTP/1.1 **RFC 7231**, Seção 6.3.6
206	PARTIAL_CONTENT	HTTP/1.1 **RFC 7233**, Seção 4.1
207	MULTI_STATUS	WebDAV **RFC 4918**, Seção 11.1
208	ALREADY_REPORTED	Extensões Vinculadas WebDAV **RFC 5842**, Seção 7.1 (Experimental)
226	IM_USED	Codificador Delta em HTTP **RFC 3229**, Seção 10.4.1
300	MULTIPLE_CHOICES	HTTP/1.1 **RFC 7231**, Seção 6.4.1
301	MOVED_PERMANENTLY	HTTP/1.1 **RFC 7231**, Seção 6.4.2
302	FOUND	HTTP/1.1 **RFC 7231**, Seção 6.4.3
303	SEE_OTHER	HTTP/1.1 **RFC 7231**, Seção 6.4.4
304	NOT_MODIFIED	HTTP/1.1 **RFC 7232**, Seção 4.1
305	USE_PROXY	HTTP/1.1 **RFC 7231**, Seção 6.4.5
307	TEMPORARY_REDIRECT	HTTP/1.1 **RFC 7231**, Seção 6.4.7
308	PERMANENT_REDIRECT	Redirecionamento Permanente **RFC 7238**, Seção 3 (Experimental)
400	BAD_REQUEST	HTTP/1.1 **RFC 7231**, Seção 6.5.1
401	UNAUTHORIZED	HTTP/1.1 Authentication **RFC 7235**, Seção 3.1
402	PAYMENT_REQUIRED	HTTP/1.1 **RFC 7231**, Seção 6.5.2
403	FORBIDDEN	HTTP/1.1 **RFC 7231**, Seção 6.5.3
404	NOT_FOUND	HTTP/1.1 **RFC 7231**, Seção 6.5.4
405	METHOD_NOT_ALLOWED	HTTP/1.1 **RFC 7231**, Seção 6.5.5
406	NOT_ACCEPTABLE	HTTP/1.1 **RFC 7231**, Seção 6.5.6
407	PROXY_AUTHENTICATION_REQUIRED	Autenticação HTTP/1.1 **RFC 7235**, Seção 3.2
408	REQUEST_TIMEOUT	HTTP/1.1 **RFC 7231**, Seção 6.5.7
409	CONFLICT	HTTP/1.1 **RFC 7231**, Seção 6.5.8
410	GONE	HTTP/1.1 **RFC 7231**, Seção 6.5.9
411	LENGTH_REQUIRED	HTTP/1.1 **RFC 7231**, Seção 6.5.10
412	PRECONDITION_FAILED	HTTP/1.1 **RFC 7232**, Seção 4.2
413	REQUEST_ENTITY_TOO_LARGE	HTTP/1.1 **RFC 7231**, Seção 6.5.11
414	REQUEST_URI_TOO_LONG	HTTP/1.1 **RFC 7231**, Seção 6.5.12
415	UNSUPPORTED_MEDIA_TYPE	HTTP/1.1 **RFC 7231**, Seção 6.5.13
416	REQUESTED_RANGE_NOT_SATISFIABLE	Alcance de Requisições HTTP/1.1 **RFC 7233**, Seção 4.4
417	EXPECTATION_FAILED	HTTP/1.1 **RFC 7231**, Seção 6.5.14
421	MISDIRECTED_REQUEST	HTTP/2 **RFC 7540**, Seção 9.1.2
422	UNPROCESSABLE_ENTITY	WebDAV **RFC 4918**, Section 11.2
423	LOCKED	WebDAV **RFC 4918**, Seção 11.3
424	FAILED_DEPENDENCY	WebDAV **RFC 4918**, Section 11.4
426	UPGRADE_REQUIRED	HTTP/1.1 **RFC 7231**, Seção 6.5.15
428	PRECONDITION_REQUIRED	Códigos de Status HTTP Adicionais **RFC 6585**
429	TOO_MANY_REQUESTS	Códigos de Status HTTP Adicionais **RFC 6585**

Continuação na pró

Tabela 1 – continuação da página anterior

Código	Nome da Enumeração	Detalhes
431	REQUEST_HEADER_FIELDS_TOO_LARGE	Códigos de Status HTTP Adicionais **RFC 6585**
500	INTERNAL_SERVER_ERROR	HTTP/1.1 **RFC 7231**, **RFC 7725** 6.6.1
501	NOT_IMPLEMENTED	HTTP/1.1 **RFC 7231**, Seção 6.6.2
502	BAD_GATEWAY	HTTP/1.1 **RFC 7231**, Seção 6.6.3
503	SERVICE_UNAVAILABLE	HTTP/1.1 **RFC 7231**, Seção 6.6.4
504	GATEWAY_TIMEOUT	HTTP/1.1 **RFC 7231**, Seção 6.6.5
505	HTTP_VERSION_NOT_SUPPORTED	HTTP/1.1 **RFC 7231**, Seção 6.6.6
506	VARIANT_ALSO_NEGOTIATES	Negociação Transparente de Conteúdo em HTTP **RFC 2295**, Seção 8.1 (E
507	INSUFFICIENT_STORAGE	WebDAV **RFC 4918**, Seção 11.5
508	LOOP_DETECTED	Extensões de Ligação WebDAV **RFC 5842**, Seção 7.2 (Experimental)
510	NOT_EXTENDED	Um Framework de Extensão HTTP **RFC 2774**, Seção 7 (Experimental)
511	NETWORK_AUTHENTICATION_REQUIRED	Códigos de Status HTTP Adicionais **RFC 6585**, Seção 6

Para preservar compatibilidades anteriores, valores de enumerações também estão presentes no módulo *http.client* na forma de constantes. O nome da enumeração é igual ao nome da constante (i.e. `http.HTTPStatus.OK` também está disponível como `http.client.OK`).

Alterado na versão 3.7: Código de status `421 MISDIRECTED_REQUEST` adicionado.

22.12 `http.client` — cliente de protocolo HTTP

Código Fonte: Lib/http/client.py

Este módulo define classes que implementam o lado do cliente nos protocolos HTTP e HTTPS. Normalmente não é utilizada diretamente – o módulo *urllib.request* o utiliza para manipular URLs que utilizam HTTP e HTTPS.

Ver também:

O Pacote de requisições é recomendado para uma interface alto-nível de cliente HTTP.

Nota: Suporte HTTPS está disponível somente se Python foi compilado com suporte SSL (através do módulo *ssl*).

O módulo fornece as seguintes classes:

class http.client.**HTTPConnection**(*host*, *port=None*[, *timeout*], *source_address=None*, *blocksize=8192*)

Uma instância *HTTPConnection* representa uma transação com um servidor HTTP. Deve ser instanciada passando a ela um host e um número de porta opcional. Se nenhum número de porta for informado, a porta é extraída da string do host se ela tem o formato `host:porta`, se não é utilizada a porta padrão HTTP (80). Se um parâmetro *timeout* opcional for dado, operações de bloqueio (como tentativas de conexão) vão expirar depois desta quantidade de segundos (se não for fornecida, as configurações padrões globais de timeout são utilizadas). O parâmetro opcional *source_adress* deve ser uma tupla com (host, porta) para utilização do endereço de origem de onde a conexão HTTP está sendo feita. O parâmetro opcional *blocksize* define o tamanho do buffer em bytes para o envio de um corpo de mensagem semelhante a um arquivo.

Por exemplo, todas as seguintes chamadas criam instâncias que conectam ao servidor com o mesmo host e porta:

```
>>> h1 = http.client.HTTPConnection('www.python.org')
>>> h2 = http.client.HTTPConnection('www.python.org:80')
>>> h3 = http.client.HTTPConnection('www.python.org', 80)
>>> h4 = http.client.HTTPConnection('www.python.org', 80, timeout=10)
```

Alterado na versão 3.2: *source_address* foi adicionado.

Alterado na versão 3.4: O argumento *strict* foi removido. "Respostas Simples" HTTP com o estilo 0.9 não são mais suportadas.

Alterado na versão 3.7: O argumento *blocksize* foi adicionado.

class http.client.**HTTPSConnection**(*host*, *port=None*, *key_file=None*, *cert_file=None*[, *timeout*], *source_address=None*, *, *context=None*, *check_hostname=None*, *blocksize=8192*)

Uma subclasse de *HTTPConnection* que utiliza SSL para comunicação com servidores seguros. A porta padrão é 443. Se *context* for especificado, ele deve ser uma instância de *ssl.SSLContext* descrevendo as várias opções do SSL.

Por favor leia *Considerações de segurança* para mais informações sobre as melhores práticas.

Alterado na versão 3.2: *source_address*, *context* e *check_hostname* foram adicionados.

Alterado na versão 3.2: Esta classe agora suporta hosts virtuais HTTPS se possível (isto é, se *ssl.HAS_SNI* é true).

Alterado na versão 3.4: O argumento *strict* foi removido. "Respostas Simples" HTTP com o estilo 0.9 não são mais suportadas.

Alterado na versão 3.4.3: Essa classe agora executa todos os certificados e verificação de hostnames necessários por padrão. Para reverter ao comportamento anterior, sem verificação, `ssl._create_unverified_context()` pode ser fornecida ao argumento *context*.

Alterado na versão 3.7.4: Esta classe agora habilita TLS 1.3 *ssl.SSLContext.post_handshake_auth* para o padrão *context* ou quanto *cert_file* é fornecido com um *context* personalizado.

Obsoleto desde a versão 3.6: *key_file* e *cert_file* estão descontinuados em favor de *context*. Por favor, em vez disso, utilize *ssl.SSLContext.load_cert_chain()* ou deixe *ssl.create_default_context()* selecionar os certificados CA confiáveis do sistema para você.

O argumento *check_hostname* também está descontinuado; o atributo *ssl.SSLContext.check_hostname* de *context* deve ser usado em seu lugar.

class http.client.**HTTPResponse**(*sock*, *debuglevel=0*, *method=None*, *url=None*)

Classe em que instâncias são retornadas mediante de conexão bem-sucedida. Não é instanciável diretamente pelo usuário.

Alterado na versão 3.4: O argumento *strict* foi removido. "Respostas Simples" HTTP com o estilo 0.9 não são mais suportadas.

As seguintes exceções são lançadas conforme apropriado:

exception http.client.**HTTPException**

A classe base das outras exceções neste módulo. É uma subclasse de *Exception*.

exception http.client.**NotConnected**

Uma subclasse de *HTTPException*.

exception http.client.**InvalidURL**

Uma subclasse de *HTTPException*, lançada se uma porta é fornecida e esta é não-numérica ou vazia.

exception http.client.**UnknownProtocol**

Uma subclasse de *HTTPException*.

exception http.client.**UnknownTransferEncoding**

Uma subclasse de *HTTPException*.

exception http.client.**UnimplementedFileMode**

Uma subclasse de *HTTPException*.

exception http.client.**IncompleteRead**
Uma subclasse de *HTTPException*.

exception http.client.**ImproperConnectionState**
Uma subclasse de *HTTPException*.

exception http.client.**CannotSendRequest**
Uma subclasse de *ImproperConnectionState*.

exception http.client.**CannotSendHeader**
Uma subclasse de *ImproperConnectionState*.

exception http.client.**ResponseNotReady**
Uma subclasse de *ImproperConnectionState*.

exception http.client.**BadStatusLine**
Uma subclasse de *HTTPException*. Lançada se um servidor responde com um código de status HTTP que não é entendido.

exception http.client.**LineTooLong**
Uma subclasse de *HTTPException*. Lançada se uma linha excessivamente longa é recebida, do servidor, no protocolo HTTP.

exception http.client.**RemoteDisconnected**
Uma subclasse de *ConnectionResetError* e *BadStatusLine*. Lançada por *HTTPConnection.getresponse()* quando a tentativa de ler a resposta resulta na não leitura dos dados pela conexão, indicando que o fim remoto fechou a conexão.

Novo na versão 3.5: Anteriormente, a exceção *BadStatusLine*`('')` foi lançada.

As constantes definidas neste módulo são:

http.client.**HTTP_PORT**
A porta padrão para o protocolo HTTP (sempre `80`).

http.client.**HTTPS_PORT**
A porta padrão para o protocolo HTTPS (sempre `443`).

http.client.**responses**
Este dicionário mapeia os códigos de status HTTP 1.1 para os nomes em W3C.

Exemplo: `http.client.responses[http.client.NOT_FOUND]` é `'Not Found'`.

Ver *códigos de status HTTP* para uma lista de códigos de status HTTP que estão disponíveis neste módulo como constantes.

22.12.1 Objetos de HTTPConnection

Instâncias *HTTPConnection* contêm os seguintes métodos:

HTTPConnection.**request**(*method*, *url*, *body=None*, *headers={}*, ***, *encode_chunked=False*)
Isto irá enviar uma requisição para o servidor utilizando o método de requisição HTTP *method* e o seletor *url*.

Se *body* é especificado, os dados especificados são mandados depois que os cabeçalhos estão prontos. Pode ser um *str*, um *bytes-like object*, um *file object* aberto, ou um iterável de *bytes*. Se *body* é uma string, ele é codificado como ISO-8859-1, o padrão para HTTP. Se é um objeto do tipo byte, os bytes são enviados como estão. Se é um file object, o conteúdo do arquivo é enviado; este objeto arquivo deve suportar pelo menos o método "read()'. Se o objeto arquivo é uma instância de *io.TextIOBase*, os dados retornados pelo método `read()` será codificado como ISO-8859-1, de outra forma os dados retornados por `read()` são enviados como estão. Se *body* é um iterável, os elementos do iterável são enviados até os mesmo se esgotar.

O argumento *headers* deve ser um mapeamento de cabeçalhos HTTP extras a serem enviados com a requisição.

If *headers* contains neither Content-Length nor Transfer-Encoding, but there is a request body, one of those header fields will be added automatically. If *body* is `None`, the Content-Length header is set to `0` for methods that expect a body (`PUT`, `POST`, and `PATCH`). If *body* is a string or a bytes-like object that is not also a *file*, the Content-Length header is set to its length. Any other type of *body* (files and iterables in general) will be chunk-encoded, and the Transfer-Encoding header will automatically be set instead of Content-Length.

The *encode_chunked* argument is only relevant if Transfer-Encoding is specified in *headers*. If *encode_chunked* is `False`, the HTTPConnection object assumes that all encoding is handled by the calling code. If it is `True`, the body will be chunk-encoded.

Nota: Chunked transfer encoding has been added to the HTTP protocol version 1.1. Unless the HTTP server is known to handle HTTP 1.1, the caller must either specify the Content-Length, or must pass a *str* or bytes-like object that is not also a file as the body representation.

Novo na versão 3.2: *body* pode agora ser um iterável.

Alterado na versão 3.6: If neither Content-Length nor Transfer-Encoding are set in *headers*, file and iterable *body* objects are now chunk-encoded. The *encode_chunked* argument was added. No attempt is made to determine the Content-Length for file objects.

`HTTPConnection.`**`getresponse`**`()`
Should be called after a request is sent to get the response from the server. Returns an *HTTPResponse* instance.

Nota: Note that you must have read the whole response before you can send a new request to the server.

Alterado na versão 3.5: Se uma *ConnectionError* ou subclasse for levantada, o objeto *HTTPConnection* estará pronto para se reconectar quando uma nova solicitação for enviada.

`HTTPConnection.`**`set_debuglevel`**`(`*level*`)`
Set the debugging level. The default debug level is `0`, meaning no debugging output is printed. Any value greater than `0` will cause all currently defined debug output to be printed to stdout. The `debuglevel` is passed to any new *HTTPResponse* objects that are created.

Novo na versão 3.1.

`HTTPConnection.`**`set_tunnel`**`(`*host*, *port=None*, *headers=None*`)`
Set the host and the port for HTTP Connect Tunnelling. This allows running the connection through a proxy server.

The host and port arguments specify the endpoint of the tunneled connection (i.e. the address included in the CONNECT request, *not* the address of the proxy server).

The headers argument should be a mapping of extra HTTP headers to send with the CONNECT request.

For example, to tunnel through a HTTPS proxy server running locally on port 8080, we would pass the address of the proxy to the *HTTPSConnection* constructor, and the address of the host that we eventually want to reach to the *set_tunnel()* method:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost", 8080)
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD","/index.html")
```

Novo na versão 3.2.

`HTTPConnection.`**`connect`**`()`
Connect to the server specified when the object was created. By default, this is called automatically when making a request if the client does not already have a connection.

HTTPConnection.**close**()
Close the connection to the server.

HTTPConnection.**blocksize**
Buffer size in bytes for sending a file-like message body.

Novo na versão 3.7.

As an alternative to using the `request()` method described above, you can also send your request step by step, by using the four functions below.

HTTPConnection.**putrequest**(*method*, *url*, *skip_host=False*, *skip_accept_encoding=False*)
This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the *method* string, the *url* string, and the HTTP version (`HTTP/1.1`). To disable automatic sending of `Host:` or `Accept-Encoding:` headers (for example to accept additional content encodings), specify *skip_host* or *skip_accept_encoding* with non-False values.

HTTPConnection.**putheader**(*header*, *argument*[, ...])
Send an **RFC 822**-style header to the server. It sends a line to the server consisting of the header, a colon and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

HTTPConnection.**endheaders**(*message_body=None*, ***, *encode_chunked=False*)
Send a blank line to the server, signalling the end of the headers. The optional *message_body* argument can be used to pass a message body associated with the request.

If *encode_chunked* is `True`, the result of each iteration of *message_body* will be chunk-encoded as specified in **RFC 7230**, Section 3.3.1. How the data is encoded is dependent on the type of *message_body*. If *message_body* implements the buffer interface the encoding will result in a single chunk. If *message_body* is a *collections.abc.Iterable*, each iteration of *message_body* will result in a chunk. If *message_body* is a *file object*, each call to `.read()` will result in a chunk. The method automatically signals the end of the chunk-encoded data immediately after *message_body*.

Nota: Due to the chunked encoding specification, empty chunks yielded by an iterator body will be ignored by the chunk-encoder. This is to avoid premature termination of the read of the request by the target server due to malformed encoding.

Novo na versão 3.6: Chunked encoding support. The *encode_chunked* parameter was added.

HTTPConnection.**send**(*data*)
Send data to the server. This should be used directly only after the *endheaders()* method has been called and before *getresponse()* is called.

22.12.2 Objetos HTTPResponse

An *HTTPResponse* instance wraps the HTTP response from the server. It provides access to the request headers and the entity body. The response is an iterable object and can be used in a with statement.

Alterado na versão 3.5: The *io.BufferedIOBase* interface is now implemented and all of its reader operations are supported.

HTTPResponse.**read**([*amt*])
Reads and returns the response body, or up to the next *amt* bytes.

HTTPResponse.**readinto**(*b*)
Reads up to the next len(b) bytes of the response body into the buffer *b*. Returns the number of bytes read.

Novo na versão 3.3.

HTTPResponse.**getheader**(*name*, *default=None*)
Return the value of the header *name*, or *default* if there is no header matching *name*. If there is more than one header with the name *name*, return all of the values joined by ', '. If 'default' is any iterable other than a single string, its elements are similarly returned joined by commas.

HTTPResponse.**getheaders**()
Return a list of (header, value) tuples.

HTTPResponse.**fileno**()
Return the `fileno` of the underlying socket.

HTTPResponse.**msg**
A `http.client.HTTPMessage` instance containing the response headers. `http.client.HTTPMessage` is a subclass of *email.message.Message*.

HTTPResponse.**version**
HTTP protocol version used by server. 10 for HTTP/1.0, 11 for HTTP/1.1.

HTTPResponse.**status**
Status code returned by server.

HTTPResponse.**reason**
Reason phrase returned by server.

HTTPResponse.**debuglevel**
A debugging hook. If *debuglevel* is greater than zero, messages will be printed to stdout as the response is read and parsed.

HTTPResponse.**closed**
Is `True` if the stream is closed.

22.12.3 Exemplos

Here is an example session that uses the `GET` method:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read()  # This will return entire content.
>>> # The following example demonstrates reading data in chunks.
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> while True:
...     chunk = r1.read(200)  # 200 bytes
...     if not chunk:
...         break
...     print(repr(chunk))
b'<!doctype html>\n<!--[if"...
...
>>> # Example of an invalid request
>>> conn = http.client.HTTPSConnection("docs.python.org")
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print(r2.status, r2.reason)
404 Not Found
```

(continua na próxima página)

(continuação da página anterior)

```
>>> data2 = r2.read()
>>> conn.close()
```

Here is an example session that uses the `HEAD` method. Note that the `HEAD` method never returns any data.

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> res = conn.getresponse()
>>> print(res.status, res.reason)
200 OK
>>> data = res.read()
>>> print(len(data))
0
>>> data == b''
True
```

Here is an example session that shows how to `POST` requests:

```
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524, '@type': 'issue', '@action':
↪'show'})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...            "Accept": "text/plain"}
>>> conn = http.client.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
b'Redirecting to <a href="http://bugs.python.org/issue12524">http://bugs.python.org/
↪issue12524</a>'
>>> conn.close()
```

Client side `HTTP PUT` requests are very similar to `POST` requests. The difference lies only the server side where HTTP server will allow resources to be created via `PUT` request. It should be noted that custom HTTP methods are also handled in *`urllib.request.Request`* by setting the appropriate method attribute. Here is an example session that shows how to send a `PUT` request using http.client:

```
>>> # This creates an HTTP message
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/file
...
>>> import http.client
>>> BODY = "***filecontents***"
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200, OK
```

22.12.4 HTTPMessage Objects

An `http.client.HTTPMessage` instance holds the headers from an HTTP response. It is implemented using the *email.message.Message* class.

22.13 `ftplib` — FTP protocol client

Código Fonte: Lib/ftplib.py

This module defines the class *FTP* and a few related items. The *FTP* class implements the client side of the FTP protocol. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other FTP servers. It is also used by the module *urllib.request* to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see Internet **RFC 959**.

Here's a sample session using the *ftplib* module:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.debian.org')     # connect to host, default port
>>> ftp.login()                     # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')               # change into "debian" directory
>>> ftp.retrlines('LIST')           # list directory contents
-rw-rw-r--    1 1176     1176         1063 Jun 15 10:18 README
...
drwxr-sr-x    5 1176     1176         4096 Dec 19  2000 pool
drwxr-sr-x    4 1176     1176         4096 Nov 17  2008 project
drwxr-xr-x    3 1176     1176         4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> with open('README', 'wb') as fp:
>>>     ftp.retrbinary('RETR README', fp.write)
'226 Transfer complete.'
>>> ftp.quit()
```

Este módulo define os seguintes itens:

class ftplib.**FTP**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *timeout=None*, *source_address=None*)

Return a new instance of the *FTP* class. When *host* is given, the method call `connect(host)` is made. When *user* is given, additionally the method call `login(user, passwd, acct)` is made (where *passwd* and *acct* default to the empty string when not given). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if is not specified, the global default timeout setting will be used). *source_address* is a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.

The *FTP* class supports the `with` statement, e.g.:

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
...
'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp          4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp            18 Jul 10  2008 Fedora
>>>
```

Alterado na versão 3.2: Suporte para a instrução `with` foi adicionado.

Alterado na versão 3.3: *source_address* parameter was added.

class ftplib.**FTP_TLS**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *keyfile=None*, *certfile=None*, *context=None*, *timeout=None*, *source_address=None*)

A *FTP* subclass which adds TLS support to FTP as described in **RFC 4217**. Connect as usual to port 21 implicitly securing the FTP control connection before authenticating. Securing the data connection requires the user to explicitly ask for it by calling the *prot_p()* method. *context* is a *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *Considerações de segurança* for best practices.

keyfile and *certfile* are a legacy alternative to *context* – they can point to PEM-formatted private key and certificate chain files (respectively) for the SSL connection.

Novo na versão 3.2.

Alterado na versão 3.3: *source_address* parameter was added.

Alterado na versão 3.4: The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

Obsoleto desde a versão 3.6: *keyfile* and *certfile* are deprecated in favor of *context*. Please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system's trusted CA certificates for you.

Here's a sample session using the *FTP_TLS* class:

```
>>> ftps = FTP_TLS('ftp.pureftpd.org')
>>> ftps.login()
'230 Anonymous user logged in'
>>> ftps.prot_p()
'200 Data protection level set to "private"'
>>> ftps.nlst()
['6jack', 'OpenBSD', 'antilink', 'blogbench', 'bsdcam', 'clockspeed', 'djbdns-jedi
↪', 'docs', 'eaccelerator-jedi', 'favicon.ico', 'francotone', 'fugu', 'ignore',
↪'libpuzzle', 'metalog', 'minidentd', 'misc', 'mysql-udf-global-user-variables',
↪'php-jenkins-hash', 'php-skein-hash', 'php-webdav', 'phpaudit', 'phpbench',
↪'pincaster', 'ping', 'posto', 'pub', 'public', 'public_keys', 'pure-ftpd',
↪'qscan', 'qtc', 'sharedance', 'skycache', 'sound', 'tmp', 'ucarp']
```

exception ftplib.**error_reply**

Exception raised when an unexpected reply is received from the server.

exception ftplib.**error_temp**

Exception raised when an error code signifying a temporary error (response codes in the range 400–499) is received.

exception ftplib.**error_perm**

Exception raised when an error code signifying a permanent error (response codes in the range 500–599) is received.

exception ftplib.**error_proto**

Exception raised when a reply is received from the server that does not fit the response specifications of the File Transfer Protocol, i.e. begin with a digit in the range 1–5.

ftplib.**all_errors**

The set of all exceptions (as a tuple) that methods of *FTP* instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed above as well as *OSError*.

Ver também:

Módulo ***netrc*** Parser for the `.netrc` file format. The file `.netrc` is typically used by FTP clients to load user authentication information before prompting the user.

22.13.1 Objetos FTP

Several methods are available in two flavors: one for handling text files and another for binary files. These are named for the command which is used followed by `lines` for the text version or `binary` for the binary version.

FTP instances have the following methods:

`FTP.`**`set_debuglevel`**(*level*)
Set the instance's debugging level. This controls the amount of debugging output printed. The default, `0`, produces no debugging output. A value of `1` produces a moderate amount of debugging output, generally a single line per request. A value of `2` or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

`FTP.`**`connect`**(*host=''*, *port=0*, *timeout=None*, *source_address=None*)
Connect to the given host and port. The default port number is `21`, as specified by the FTP protocol specification. It is rarely needed to specify a different port number. This function should be called only once for each instance; it should not be called at all if a host was given when the instance was created. All other methods can only be used after a connection has been made. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If no *timeout* is passed, the global default timeout setting will be used. *source_address* is a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.

Alterado na versão 3.3: *source_address* parameter was added.

`FTP.`**`getwelcome`**()
Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

`FTP.`**`login`**(*user='anonymous'*, *passwd=''*, *acct=''*)
Log in as the given *user*. The *passwd* and *acct* parameters are optional and default to the empty string. If no *user* is specified, it defaults to `'anonymous'`. If *user* is `'anonymous'`, the default *passwd* is `'anonymous@'`. This function should be called only once for each instance, after a connection has been established; it should not be called at all if a host and user were given when the instance was created. Most FTP commands are only allowed after the client has logged in. The *acct* parameter supplies "accounting information"; few systems implement this.

`FTP.`**`abort`**()
Abort a file transfer that is in progress. Using this does not always work, but it's worth a try.

`FTP.`**`sendcmd`**(*cmd*)
Send a simple command string to the server and return the response string.

`FTP.`**`voidcmd`**(*cmd*)
Send a simple command string to the server and handle the response. Return nothing if a response code corresponding to success (codes in the range 200–299) is received. Raise *error_reply* otherwise.

`FTP.`**`retrbinary`**(*cmd*, *callback*, *blocksize=8192*, *rest=None*)
Retrieve a file in binary transfer mode. *cmd* should be an appropriate `RETR` command: `'RETR filename'`. The *callback* function is called for each block of data received, with a single bytes argument giving the data block. The optional *blocksize* argument specifies the maximum chunk size to read on the low-level socket object created to do the actual transfer (which will also be the largest size of the data blocks passed to *callback*). A reasonable default is chosen. *rest* means the same thing as in the *transfercmd()* method.

`FTP.`**`retrlines`**(*cmd*, *callback=None*)
Retrieve a file or directory listing in ASCII transfer mode. *cmd* should be an appropriate `RETR` command (see *retrbinary()*) or a command such as `LIST` or `NLST` (usually just the string `'LIST'`). `LIST` retrieves a list of files and information about those files. `NLST` retrieves a list of file names. The *callback* function is called for

each line with a string argument containing the line with the trailing CRLF stripped. The default *callback* prints the line to `sys.stdout`.

`FTP.`**`set_pasv`**(*val*)
Enable "passive" mode if *val* is true, otherwise disable passive mode. Passive mode is on by default.

`FTP.`**`storbinary`**(*cmd*, *fp*, *blocksize=8192*, *callback=None*, *rest=None*)
Store a file in binary transfer mode. *cmd* should be an appropriate `STOR` command: `"STOR filename"`. *fp* is a *file object* (opened in binary mode) which is read until EOF using its `read()` method in blocks of size *blocksize* to provide the data to be stored. The *blocksize* argument defaults to 8192. *callback* is an optional single parameter callable that is called on each block of data after it is sent. *rest* means the same thing as in the *transfercmd()* method.

Alterado na versão 3.2: Parâmetro *rest* adicionado.

`FTP.`**`storlines`**(*cmd*, *fp*, *callback=None*)
Store a file in ASCII transfer mode. *cmd* should be an appropriate `STOR` command (see *storbinary()*). Lines are read until EOF from the *file object* *fp* (opened in binary mode) using its *readline()* method to provide the data to be stored. *callback* is an optional single parameter callable that is called on each line after it is sent.

`FTP.`**`transfercmd`**(*cmd*, *rest=None*)
Initiate a transfer over the data connection. If the transfer is active, send an `EPRT` or `PORT` command and the transfer command specified by *cmd*, and accept the connection. If the server is passive, send an `EPSV` or `PASV` command, connect to it, and start the transfer command. Either way, return the socket for the connection.

If optional *rest* is given, a `REST` command is sent to the server, passing *rest* as an argument. *rest* is usually a byte offset into the requested file, telling the server to restart sending the file's bytes at the requested offset, skipping over the initial bytes. Note however that **RFC 959** requires only that *rest* be a string containing characters in the printable range from ASCII code 33 to ASCII code 126. The *transfercmd()* method, therefore, converts *rest* to a string, but no check is performed on the string's contents. If the server does not recognize the `REST` command, an *error_reply* exception will be raised. If this happens, simply call *transfercmd()* without a *rest* argument.

`FTP.`**`ntransfercmd`**(*cmd*, *rest=None*)
Like *transfercmd()*, but returns a tuple of the data connection and the expected size of the data. If the expected size could not be computed, `None` will be returned as the expected size. *cmd* and *rest* means the same thing as in *transfercmd()*.

`FTP.`**`mlsd`**(*path=""*, *facts=[]*)
List a directory in a standardized format by using `MLSD` command (**RFC 3659**). If *path* is omitted the current directory is assumed. *facts* is a list of strings representing the type of information desired (e.g. `["type", "size", "perm"]`). Return a generator object yielding a tuple of two elements for every file found in path. First element is the file name, the second one is a dictionary containing facts about the file name. Content of this dictionary might be limited by the *facts* argument but server is not guaranteed to return all requested facts.

Novo na versão 3.3.

`FTP.`**`nlst`**(*argument*[, ...])
Return a list of file names as returned by the `NLST` command. The optional *argument* is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `NLST` command.

Nota: If your server supports the command, *mlsd()* offers a better API.

`FTP.`**`dir`**(*argument*[, ...])
Produce a directory listing as returned by the `LIST` command, printing it to standard output. The optional *argument* is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `LIST` command. If the last argument is a function, it is used as a *callback* function as for *retrlines()*; the default prints to `sys.stdout`. This method returns `None`.

Nota: If your server supports the command, `mlsd()` offers a better API.

FTP.**rename**(*fromname*, *toname*)
Rename file *fromname* on the server to *toname*.

FTP.**delete**(*filename*)
Remove the file named *filename* from the server. If successful, returns the text of the response, otherwise raises `error_perm` on permission errors or `error_reply` on other errors.

FTP.**cwd**(*pathname*)
Set the current directory on the server.

FTP.**mkd**(*pathname*)
Create a new directory on the server.

FTP.**pwd**()
Return the pathname of the current directory on the server.

FTP.**rmd**(*dirname*)
Remove the directory named *dirname* on the server.

FTP.**size**(*filename*)
Request the size of the file named *filename* on the server. On success, the size of the file is returned as an integer, otherwise `None` is returned. Note that the `SIZE` command is not standardized, but is supported by many common server implementations.

FTP.**quit**()
Send a `QUIT` command to the server and close the connection. This is the "polite" way to close a connection, but it may raise an exception if the server responds with an error to the `QUIT` command. This implies a call to the `close()` method which renders the `FTP` instance useless for subsequent calls (see below).

FTP.**close**()
Close the connection unilaterally. This should not be applied to an already closed connection such as after a successful call to `quit()`. After this call the `FTP` instance should not be used any more (after a call to `close()` or `quit()` you cannot reopen the connection by issuing another `login()` method).

22.13.2 Objetos FTP_TLS

`FTP_TLS` class inherits from `FTP`, defining these additional objects:

FTP_TLS.**ssl_version**
The SSL version to use (defaults to `ssl.PROTOCOL_SSLv23`).

FTP_TLS.**auth**()
Set up a secure control connection by using TLS or SSL, depending on what is specified in the `ssl_version` attribute.

Alterado na versão 3.4: The method now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

FTP_TLS.**ccc**()
Revert control channel back to plaintext. This can be useful to take advantage of firewalls that know how to handle NAT with non-secure FTP without opening fixed ports.

Novo na versão 3.3.

FTP_TLS.**prot_p**()
Set up secure data connection.

FTP_TLS.**prot_c**()
Set up clear text data connection.

22.14 poplib — POP3 protocol client

Código Fonte: Lib/poplib.py

This module defines a class, *POP3*, which encapsulates a connection to a POP3 server and implements the protocol as defined in **RFC 1939**. The *POP3* class supports both the minimal and optional command sets from **RFC 1939**. The *POP3* class also supports the STLS command introduced in **RFC 2595** to enable encrypted communication on an already established connection.

Additionally, this module provides a class *POP3_SSL*, which provides support for connecting to POP3 servers that use SSL as an underlying protocol layer.

Note that POP3, though widely supported, is obsolescent. The implementation quality of POP3 servers varies widely, and too many are quite poor. If your mailserver supports IMAP, you would be better off using the *imaplib.IMAP4* class, as IMAP servers tend to be better implemented.

The *poplib* module provides two classes:

class poplib.**POP3**(*host*, *port=POP3_PORT*[, *timeout*])
This class implements the actual POP3 protocol. The connection is created when the instance is initialized. If *port* is omitted, the standard POP3 port (110) is used. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt (if not specified, the global default timeout setting will be used).

class poplib.**POP3_SSL**(*host*, *port=POP3_SSL_PORT*, *keyfile=None*, *certfile=None*, *timeout=None*, *context=None*)
This is a subclass of *POP3* that connects to the server over an SSL encrypted socket. If *port* is not specified, 995, the standard POP3-over-SSL port is used. *timeout* works as in the *POP3* constructor. *context* is an optional *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *Considerações de segurança* for best practices.

keyfile and *certfile* are a legacy alternative to *context* - they can point to PEM-formatted private key and certificate chain files, respectively, for the SSL connection.

Alterado na versão 3.2: Parâmetro *context* adicionado.

Alterado na versão 3.4: The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

Obsoleto desde a versão 3.6: *keyfile* and *certfile* are deprecated in favor of *context*. Please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system's trusted CA certificates for you.

One exception is defined as an attribute of the *poplib* module:

exception poplib.**error_proto**
Exception raised on any errors from this module (errors from *socket* module are not caught). The reason for the exception is passed to the constructor as a string.

Ver também:

Módulo *imaplib* The standard Python IMAP module.

Frequently Asked Questions About Fetchmail The FAQ for the **fetchmail** POP/IMAP client collects information on POP3 server variations and RFC noncompliance that may be useful if you need to write an application based on the POP protocol.

22.14.1 Objetos POP3

All POP3 commands are represented by methods of the same name, in lower-case; most return the response text sent by the server.

An *POP3* instance has the following methods:

POP3.**set_debuglevel**(*level*)
Set the instance's debugging level. This controls the amount of debugging output printed. The default, `0`, produces no debugging output. A value of `1` produces a moderate amount of debugging output, generally a single line per request. A value of `2` or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

POP3.**getwelcome**()
Returns the greeting string sent by the POP3 server.

POP3.**capa**()
Query the server's capabilities as specified in **RFC 2449**. Returns a dictionary in the form `{'name': ['param'...]}`.

Novo na versão 3.4.

POP3.**user**(*username*)
Send user command, response should indicate that a password is required.

POP3.**pass_**(*password*)
Send password, response includes message count and mailbox size. Note: the mailbox on the server is locked until `quit()` is called.

POP3.**apop**(*user*, *secret*)
Use the more secure APOP authentication to log into the POP3 server.

POP3.**rpop**(*user*)
Use RPOP authentication (similar to UNIX r-commands) to log into POP3 server.

POP3.**stat**()
Get mailbox status. The result is a tuple of 2 integers: `(message count, mailbox size)`.

POP3.**list**([*which*])
Request message list, result is in the form `(response, ['mesg_num octets', ...], octets)`. If *which* is set, it is the message to list.

POP3.**retr**(*which*)
Retrieve whole message number *which*, and set its seen flag. Result is in form `(response, ['line', ...], octets)`.

POP3.**dele**(*which*)
Flag message number *which* for deletion. On most servers deletions are not actually performed until QUIT (the major exception is Eudora QPOP, which deliberately violates the RFCs by doing pending deletes on any disconnect).

POP3.**rset**()
Remove any deletion marks for the mailbox.

POP3.**noop**()
Do nothing. Might be used as a keep-alive.

POP3.**quit**()
Signoff: commit changes, unlock mailbox, drop connection.

POP3.**top**(*which*, *howmuch*)
Retrieves the message header plus *howmuch* lines of the message after the header of message number *which*. Result is in form `(response, ['line', ...], octets)`.

The POP3 TOP command this method uses, unlike the RETR command, doesn't set the message's seen flag; unfortunately, TOP is poorly specified in the RFCs and is frequently broken in off-brand servers. Test this method by hand against the POP3 servers you will use before trusting it.

POP3.**uidl**(*which=None*)
Return message digest (unique id) list. If *which* is specified, result contains the unique id for that message in the form `'response mesgnum uid`, otherwise result is list `(response, ['mesgnum uid', ...], octets)`.

POP3.**utf8**()
Try to switch to UTF-8 mode. Returns the server response if successful, raises *error_proto* if not. Specified in **RFC 6856**.

Novo na versão 3.5.

POP3.**stls**(*context=None*)
Start a TLS session on the active connection as specified in **RFC 2595**. This is only allowed before user authentication

context parameter is a *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *Considerações de segurança* for best practices.

This method supports hostname checking via *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

Novo na versão 3.4.

Instances of *POP3_SSL* have no additional methods. The interface of this subclass is identical to its parent.

22.14.2 Exemplo POP3

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print(j)
```

At the end of the module, there is a test section that contains a more extensive example of usage.

22.15 `imaplib` — IMAP4 protocol client

Código Fonte: Lib/imaplib.py

This module defines three classes, *IMAP4*, *IMAP4_SSL* and *IMAP4_stream*, which encapsulate a connection to an IMAP4 server and implement a large subset of the IMAP4rev1 client protocol as defined in **RFC 2060**. It is backward compatible with IMAP4 (**RFC 1730**) servers, but note that the `STATUS` command is not supported in IMAP4.

Three classes are provided by the *imaplib* module, *IMAP4* is the base class:

class imaplib.**IMAP4**(*host=''*, *port=IMAP4_PORT*)

This class implements the actual IMAP4 protocol. The connection is created and protocol version (IMAP4 or IMAP4rev1) is determined when the instance is initialized. If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4 port (143) is used.

The *IMAP4* class supports the `with` statement. When used like this, the IMAP4 `LOGOUT` command is issued automatically when the `with` statement exits. E.g.:

```
>>> from imaplib import IMAP4
>>> with IMAP4("domain.org") as M:
...     M.noop()
...
('OK', [b'Nothing Accomplished. d25if65hy903weo.87'])
```

Alterado na versão 3.5: Suporte para a instrução `with` foi adicionado.

Three exceptions are defined as attributes of the *IMAP4* class:

exception IMAP4.**error**

Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

exception IMAP4.**abort**

IMAP4 server errors cause this exception to be raised. This is a sub-class of *IMAP4.error*. Note that closing the instance and instantiating a new one will usually allow recovery from this exception.

exception IMAP4.**readonly**

This exception is raised when a writable mailbox has its status changed by the server. This is a sub-class of *IMAP4.error*. Some other client now has write permission, and the mailbox will need to be re-opened to re-obtain write permission.

There's also a subclass for secure connections:

class imaplib.**IMAP4_SSL**(*host=''*, *port=IMAP4_SSL_PORT*, *keyfile=None*, *certfile=None*, *ssl_context=None*)

This is a subclass derived from *IMAP4* that connects over an SSL encrypted socket (to use this class you need a socket module that was compiled with SSL support). If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4-over-SSL port (993) is used. *ssl_context* is a *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *Considerações de segurança* for best practices.

keyfile and *certfile* are a legacy alternative to *ssl_context* - they can point to PEM-formatted private key and certificate chain files for the SSL connection. Note that the *keyfile*/*certfile* parameters are mutually exclusive with *ssl_context*, a *ValueError* is raised if *keyfile*/*certfile* is provided along with *ssl_context*.

Alterado na versão 3.3: *ssl_context* parameter added.

Alterado na versão 3.4: The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

Obsoleto desde a versão 3.6: *keyfile* and *certfile* are deprecated in favor of *ssl_context*. Please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system's trusted CA certificates for you.

The second subclass allows for connections created by a child process:

class imaplib.**IMAP4_stream**(*command*)

This is a subclass derived from *IMAP4* that connects to the `stdin/stdout` file descriptors created by passing *command* to `subprocess.Popen()`.

The following utility functions are defined:

imaplib.**Internaldate2tuple**(*datestr*)
Parse an IMAP4 INTERNALDATE string and return corresponding local time. The return value is a *time.struct_time* tuple or None if the string has wrong format.

imaplib.**Int2AP**(*num*)
Converts an integer into a string representation using characters from the set [A .. P].

imaplib.**ParseFlags**(*flagstr*)
Converts an IMAP4 FLAGS response to a tuple of individual flags.

imaplib.**Time2Internaldate**(*date_time*)
Convert *date_time* to an IMAP4 INTERNALDATE representation. The return value is a string in the form: "DD-Mmm-YYYY HH:MM:SS +HHMM" (including double-quotes). The *date_time* argument can be a number (int or float) representing seconds since epoch (as returned by *time.time()*), a 9-tuple representing local time an instance of *time.struct_time* (as returned by *time.localtime()*), an aware instance of *datetime.datetime*, or a double-quoted string. In the last case, it is assumed to already be in the correct format.

Note that IMAP4 message numbers change as the mailbox changes; in particular, after an EXPUNGE command performs deletions the remaining messages are renumbered. So it is highly advisable to use UIDs instead, with the UID command.

At the end of the module, there is a test section that contains a more extensive example of usage.

Ver também:

Documents describing the protocol, and sources and binaries for servers implementing it, can all be found at the University of Washington's *IMAP Information Center* (https://www.washington.edu/imap/).

22.15.1 Objetos IMAP4

All IMAP4rev1 commands are represented by methods of the same name, either upper-case or lower-case.

All arguments to commands are converted to strings, except for AUTHENTICATE, and the last argument to APPEND which is passed as an IMAP4 literal. If necessary (the string contains IMAP4 protocol-sensitive characters and isn't enclosed with either parentheses or double quotes) each string is quoted. However, the *password* argument to the LOGIN command is always quoted. If you want to avoid having an argument string quoted (eg: the *flags* argument to STORE) then enclose the string in parentheses (eg: r'(\Deleted)').

Each command returns a tuple: (type, [data, ...]) where *type* is usually 'OK' or 'NO', and *data* is either the text from the command response, or mandated results from the command. Each *data* is either a string, or a tuple. If a tuple, then the first part is the header of the response, and the second part contains the data (ie: 'literal' value).

The *message_set* options to commands below is a string specifying one or more messages to be acted upon. It may be a simple message number ('1'), a range of message numbers ('2:4'), or a group of non-contiguous ranges separated by commas ('1:3,6:9'). A range can contain an asterisk to indicate an infinite upper bound ('3:*').

An *IMAP4* instance has the following methods:

IMAP4.**append**(*mailbox*, *flags*, *date_time*, *message*)
Append *message* to named mailbox.

IMAP4.**authenticate**(*mechanism*, *authobject*)
Authenticate command — requires response processing.

mechanism specifies which authentication mechanism is to be used - it should appear in the instance variable capabilities in the form AUTH=mechanism.

authobject must be a callable object:

```
data = authobject(response)
```

It will be called to process server continuation responses; the *response* argument it is passed will be `bytes`. It should return `bytes` *data* that will be base64 encoded and sent to the server. It should return `None` if the client abort response `*` should be sent instead.

Alterado na versão 3.5: string usernames and passwords are now encoded to `utf-8` instead of being limited to ASCII.

`IMAP4.`**`check`**`()`
Checkpoint mailbox on server.

`IMAP4.`**`close`**`()`
Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before `LOGOUT`.

`IMAP4.`**`copy`**`(`*message_set*, *new_mailbox*`)`
Copy *message_set* messages onto end of *new_mailbox*.

`IMAP4.`**`create`**`(`*mailbox*`)`
Create new mailbox named *mailbox*.

`IMAP4.`**`delete`**`(`*mailbox*`)`
Delete old mailbox named *mailbox*.

`IMAP4.`**`deleteacl`**`(`*mailbox*, *who*`)`
Delete the ACLs (remove any rights) set for who on mailbox.

`IMAP4.`**`enable`**`(`*capability*`)`
Enable *capability* (see **RFC 5161**). Most capabilities do not need to be enabled. Currently only the `UTF8=ACCEPT` capability is supported (see **RFC 6855**).

Novo na versão 3.5: The *`enable()`* method itself, and **RFC 6855** support.

`IMAP4.`**`expunge`**`()`
Permanently remove deleted items from selected mailbox. Generates an `EXPUNGE` response for each deleted message. Returned data contains a list of `EXPUNGE` message numbers in order received.

`IMAP4.`**`fetch`**`(`*message_set*, *message_parts*`)`
Fetch (parts of) messages. *message_parts* should be a string of message part names enclosed within parentheses, eg: `"(UID BODY[TEXT])"`. Returned data are tuples of message part envelope and data.

`IMAP4.`**`getacl`**`(`*mailbox*`)`
Get the `ACL`s for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

`IMAP4.`**`getannotation`**`(`*mailbox*, *entry*, *attribute*`)`
Retrieve the specified `ANNOTATION`s for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

`IMAP4.`**`getquota`**`(`*root*`)`
Get the `quota` *root*'s resource usage and limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

`IMAP4.`**`getquotaroot`**`(`*mailbox*`)`
Get the list of `quota roots` for the named *mailbox*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

`IMAP4.`**`list`**`(`[*directory*[, *pattern*]])
List mailbox names in *directory* matching *pattern*. *directory* defaults to the top-level mail folder, and *pattern* defaults to match anything. Returned data contains a list of `LIST` responses.

`IMAP4.`**`login`**`(`*user*, *password*`)`
Identify the client using a plaintext password. The *password* will be quoted.

IMAP4.**login_cram_md5**(*user*, *password*)
Force use of CRAM-MD5 authentication when identifying the client to protect the password. Will only work if the server CAPABILITY response includes the phrase AUTH=CRAM-MD5.

IMAP4.**logout**()
Shutdown connection to server. Returns server BYE response.

IMAP4.**lsub**(*directory=""*, *pattern='*'*)
List subscribed mailbox names in directory matching pattern. *directory* defaults to the top level directory and *pattern* defaults to match any mailbox. Returned data are tuples of message part envelope and data.

IMAP4.**myrights**(*mailbox*)
Show my ACLs for a mailbox (i.e. the rights that I have on mailbox).

IMAP4.**namespace**()
Returns IMAP namespaces as defined in **RFC 2342**.

IMAP4.**noop**()
Envia NOOP para o servidor.

IMAP4.**open**(*host*, *port*)
Opens socket to *port* at *host*. This method is implicitly called by the *IMAP4* constructor. The connection objects established by this method will be used in the *IMAP4.read()*, *IMAP4.readline()*, *IMAP4.send()*, and *IMAP4.shutdown()* methods. You may override this method.

IMAP4.**partial**(*message_num*, *message_part*, *start*, *length*)
Fetch truncated part of a message. Returned data is a tuple of message part envelope and data.

IMAP4.**proxyauth**(*user*)
Assume authentication as *user*. Allows an authorised administrator to proxy into any user's mailbox.

IMAP4.**read**(*size*)
Reads *size* bytes from the remote server. You may override this method.

IMAP4.**readline**()
Reads one line from the remote server. You may override this method.

IMAP4.**recent**()
Prompt server for an update. Returned data is None if no new messages, else value of RECENT response.

IMAP4.**rename**(*oldmailbox*, *newmailbox*)
Rename mailbox named *oldmailbox* to *newmailbox*.

IMAP4.**response**(*code*)
Return data for response *code* if received, or None. Returns the given code, instead of the usual type.

IMAP4.**search**(*charset*, *criterion*[, ...])
Search mailbox for matching messages. *charset* may be None, in which case no CHARSET will be specified in the request to the server. The IMAP protocol requires that at least one criterion be specified; an exception will be raised when the server returns an error. *charset* must be None if the UTF8=ACCEPT capability was enabled using the *enable()* command.

Exemplo:

```
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

IMAP4.**select**(*mailbox='INBOX'*, *readonly=False*)
Select a mailbox. Returned data is the count of messages in *mailbox* (EXISTS response). The default *mailbox* is 'INBOX'. If the *readonly* flag is set, modifications to the mailbox are not allowed.

IMAP4.**send**(*data*)
Sends data to the remote server. You may override this method.

IMAP4.**setacl**(*mailbox*, *who*, *what*)
Set an ACL for *mailbox*. The method is non-standard, but is supported by the Cyrus server.

IMAP4.**setannotation**(*mailbox*, *entry*, *attribute*[, ...])
Set ANNOTATIONs for *mailbox*. The method is non-standard, but is supported by the Cyrus server.

IMAP4.**setquota**(*root*, *limits*)
Set the quota *root*'s resource *limits*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**shutdown**()
Close connection established in open. This method is implicitly called by *IMAP4.logout()*. You may override this method.

IMAP4.**socket**()
Returns socket instance used to connect to server.

IMAP4.**sort**(*sort_criteria*, *charset*, *search_criterion*[, ...])
The sort command is a variant of search with sorting semantics for the results. Returned data contains a space separated list of matching message numbers.

Sort has two arguments before the *search_criterion* argument(s); a parenthesized list of *sort_criteria*, and the searching *charset*. Note that unlike search, the searching *charset* argument is mandatory. There is also a uid sort command which corresponds to sort the way that uid search corresponds to search. The sort command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the numbers of matching messages.

This is an IMAP4rev1 extension command.

IMAP4.**starttls**(*ssl_context=None*)
Send a STARTTLS command. The *ssl_context* argument is optional and should be a *ssl.SSLContext* object. This will enable encryption on the IMAP connection. Please read *Considerações de segurança* for best practices.

Novo na versão 3.2.

Alterado na versão 3.4: The method now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

IMAP4.**status**(*mailbox*, *names*)
Request named status conditions for *mailbox*.

IMAP4.**store**(*message_set*, *command*, *flag_list*)
Alters flag dispositions for messages in mailbox. *command* is specified by section 6.4.6 of **RFC 2060** as being one of "FLAGS", "+FLAGS", or "-FLAGS", optionally with a suffix of ".SILENT".

For example, to set the delete flag on all messages:

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
   M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

Nota: Creating flags containing ']' (for example: "[test]") violates **RFC 3501** (the IMAP protocol). However, imaplib has historically allowed creation of such tags, and popular IMAP servers, such as Gmail, accept and produce such flags. There are non-Python programs which also create such tags. Although it is an RFC violation and IMAP

clients and servers are supposed to be strict, imaplib nonetheless continues to allow such tags to be created for backward compatibility reasons, and as of Python 3.6, handles them if they are sent from the server, since this improves real-world compatibility.

IMAP4.**subscribe**(*mailbox*)
Subscribe to new mailbox.

IMAP4.**thread**(*threading_algorithm*, *charset*, *search_criterion*[, ...])
The `thread` command is a variant of `search` with threading semantics for the results. Returned data contains a space separated list of thread members.

Thread members consist of zero or more messages numbers, delimited by spaces, indicating successive parent and child.

Thread has two arguments before the *search_criterion* argument(s); a *threading_algorithm*, and the searching *charset*. Note that unlike `search`, the searching *charset* argument is mandatory. There is also a `uid thread` command which corresponds to `thread` the way that `uid search` corresponds to `search`. The `thread` command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the matching messages threaded according to the specified threading algorithm.

This is an `IMAP4rev1` extension command.

IMAP4.**uid**(*command*, *arg*[, ...])
Execute command args with messages identified by UID, rather than message number. Returns response appropriate to command. At least one argument must be supplied; if none are provided, the server will return an error and an exception will be raised.

IMAP4.**unsubscribe**(*mailbox*)
Unsubscribe from old mailbox.

IMAP4.**xatom**(*name*[, ...])
Allow simple extension commands notified by server in `CAPABILITY` response.

The following attributes are defined on instances of *IMAP4*:

IMAP4.**PROTOCOL_VERSION**
The most recent supported protocol in the `CAPABILITY` response from the server.

IMAP4.**debug**
Integer value to control debugging output. The initialize value is taken from the module variable `Debug`. Values greater than three trace each command.

IMAP4.**utf8_enabled**
Boolean value that is normally `False`, but is set to `True` if an *enable()* command is successfully issued for the `UTF8=ACCEPT` capability.

Novo na versão 3.5.

22.15.2 Exemplo IMAP4

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
import getpass, imaplib

M = imaplib.IMAP4()
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print('Message %s\n%s\n' % (num, data[0][1]))
M.close()
M.logout()
```

22.16 `nntplib` — NNTP protocol client

Código Fonte: Lib/nntplib.py

This module defines the class *NNTP* which implements the client side of the Network News Transfer Protocol. It can be used to implement a news reader or poster, or automated news processors. It is compatible with **RFC 3977** as well as the older **RFC 977** and **RFC 2980**.

Here are two small examples of how it can be used. To list some statistics about a newsgroup and print the subjects of the last 10 articles:

```
>>> s = nntplib.NNTP('news.gmane.io')
>>> resp, count, first, last, name = s.group('gmane.comp.python.committers')
>>> print('Group', name, 'has', count, 'articles, range', first, 'to', last)
Group gmane.comp.python.committers has 1096 articles, range 1 to 1096
>>> resp, overviews = s.over((last - 9, last))
>>> for id, over in overviews:
...     print(id, nntplib.decode_header(over['subject']))
...
1087 Re: Commit privileges for Łukasz Langa
1088 Re: 3.2 alpha 2 freeze
1089 Re: 3.2 alpha 2 freeze
1090 Re: Commit privileges for Łukasz Langa
1091 Re: Commit privileges for Łukasz Langa
1092 Updated ssh key
1093 Re: Updated ssh key
1094 Re: Updated ssh key
1095 Hello fellow committers!
1096 Re: Hello fellow committers!
>>> s.quit()
'205 Bye!'
```

To post an article from a binary file (this assumes that the article has valid headers, and that you have right to post on the particular newsgroup):

```
>>> s = nntplib.NNTP('news.gmane.io')
>>> f = open('article.txt', 'rb')
>>> s.post(f)
```

(continua na próxima página)

(continuação da página anterior)

```
'240 Article posted successfully.'
>>> s.quit()
'205 Bye!'
```

The module itself defines the following classes:

class nntplib.**NNTP**(*host*, *port=119*, *user=None*, *password=None*, *readermode=None*, *usenetrc=False*[, *timeout*])

Return a new *NNTP* object, representing a connection to the NNTP server running on host *host*, listening at port *port*. An optional *timeout* can be specified for the socket connection. If the optional *user* and *password* are provided, or if suitable credentials are present in `/.netrc` and the optional flag *usenetrc* is true, the `AUTHINFO USER` and `AUTHINFO PASS` commands are used to identify and authenticate the user to the server. If the optional flag *readermode* is true, then a `mode reader` command is sent before authentication is performed. Reader mode is sometimes necessary if you are connecting to an NNTP server on the local machine and intend to call reader-specific commands, such as `group`. If you get unexpected *NNTPPermanentError*s, you might need to set *readermode*. The *NNTP* class supports the `with` statement to unconditionally consume *OSError* exceptions and to close the NNTP connection when done, e.g.:

```
>>> from nntplib import NNTP
>>> with NNTP('news.gmane.io') as n:
...     n.group('gmane.comp.python.committers')
...
('211 1755 1 1755 gmane.comp.python.committers', 1755, 1, 1755, 'gmane.comp.
↪python.committers')
>>>
```

Alterado na versão 3.2: *usenetrc* is now `False` by default.

Alterado na versão 3.3: Suporte para a instrução `with` foi adicionado.

class nntplib.**NNTP_SSL**(*host*, *port=563*, *user=None*, *password=None*, *ssl_context=None*, *readermode=None*, *usenetrc=False*[, *timeout*])

Return a new *NNTP_SSL* object, representing an encrypted connection to the NNTP server running on host *host*, listening at port *port*. *NNTP_SSL* objects have the same methods as *NNTP* objects. If *port* is omitted, port 563 (NNTPS) is used. *ssl_context* is also optional, and is a *SSLContext* object. Please read *Considerações de segurança* for best practices. All other parameters behave the same as for *NNTP*.

Note that SSL-on-563 is discouraged per **RFC 4642**, in favor of STARTTLS as described below. However, some servers only support the former.

Novo na versão 3.2.

Alterado na versão 3.4: The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

exception nntplib.**NNTPError**

Derived from the standard exception *Exception*, this is the base class for all exceptions raised by the *nntplib* module. Instances of this class have the following attribute:

response

The response of the server if available, as a *str* object.

exception nntplib.**NNTPReplyError**

Exception raised when an unexpected reply is received from the server.

exception nntplib.**NNTPTemporaryError**

Exception raised when a response code in the range 400–499 is received.

exception nntplib.**NNTPPermanentError**

Exception raised when a response code in the range 500–599 is received.

exception nntplib.**NNTPProtocolError**
Exception raised when a reply is received from the server that does not begin with a digit in the range 1–5.

exception nntplib.**NNTPDataError**
Exception raised when there is some error in the response data.

22.16.1 NNTP Objects

When connected, *NNTP* and *NNTP_SSL* objects support the following methods and attributes.

Attributes

NNTP.**nntp_version**
An integer representing the version of the NNTP protocol supported by the server. In practice, this should be 2 for servers advertising **RFC 3977** compliance and 1 for others.

Novo na versão 3.2.

NNTP.**nntp_implementation**
A string describing the software name and version of the NNTP server, or *None* if not advertised by the server.

Novo na versão 3.2.

Métodos

The *response* that is returned as the first item in the return tuple of almost all methods is the server's response: a string beginning with a three-digit code. If the server's response indicates an error, the method raises one of the above exceptions.

Many of the following methods take an optional keyword-only argument *file*. When the *file* argument is supplied, it must be either a *file object* opened for binary writing, or the name of an on-disk file to be written to. The method will then write any data returned by the server (except for the response line and the terminating dot) to the file; any list of lines, tuples or objects that the method normally returns will be empty.

Alterado na versão 3.2: Many of the following methods have been reworked and fixed, which makes them incompatible with their 3.1 counterparts.

NNTP.**quit**()
Send a QUIT command and close the connection. Once this method has been called, no other methods of the NNTP object should be called.

NNTP.**getwelcome**()
Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

NNTP.**getcapabilities**()
Return the **RFC 3977** capabilities advertised by the server, as a *dict* instance mapping capability names to (possibly empty) lists of values. On legacy servers which don't understand the CAPABILITIES command, an empty dictionary is returned instead.

```
>>> s = NNTP('news.gmane.io')
>>> 'POST' in s.getcapabilities()
True
```

Novo na versão 3.2.

NNTP.**login**(*user=None*, *password=None*, *usenetrc=True*)
Send AUTHINFO commands with the user name and password. If *user* and *password* are None and *usenetrc* is true, credentials from ~/.netrc will be used if possible.

Unless intentionally delayed, login is normally performed during the *NNTP* object initialization and separately calling this function is unnecessary. To force authentication to be delayed, you must not set *user* or *password* when creating the object, and must set *usenetrc* to False.

Novo na versão 3.2.

NNTP.**starttls**(*context=None*)
Send a STARTTLS command. This will enable encryption on the NNTP connection. The *context* argument is optional and should be a *ssl.SSLContext* object. Please read *Considerações de segurança* for best practices.

Note that this may not be done after authentication information has been transmitted, and authentication occurs by default if possible during a *NNTP* object initialization. See *NNTP.login()* for information on suppressing this behavior.

Novo na versão 3.2.

Alterado na versão 3.4: The method now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

NNTP.**newgroups**(*date*, ***, *file=None*)
Send a NEWGROUPS command. The *date* argument should be a *datetime.date* or *datetime.datetime* object. Return a pair (response, groups) where *groups* is a list representing the groups that are new since the given *date*. If *file* is supplied, though, then *groups* will be empty.

```
>>> from datetime import date, timedelta
>>> resp, groups = s.newgroups(date.today() - timedelta(days=3))
>>> len(groups)
85
>>> groups[0]
GroupInfo(group='gmane.network.tor.devel', last='4', first='1', flag='m')
```

NNTP.**newnews**(*group*, *date*, ***, *file=None*)
Send a NEWNEWS command. Here, *group* is a group name or '*', and *date* has the same meaning as for *newgroups()*. Return a pair (response, articles) where *articles* is a list of message ids.

This command is frequently disabled by NNTP server administrators.

NNTP.**list**(*group_pattern=None*, ***, *file=None*)
Send a LIST or LIST ACTIVE command. Return a pair (response, list) where *list* is a list of tuples representing all the groups available from this NNTP server, optionally matching the pattern string *group_pattern*. Each tuple has the form (group, last, first, flag), where *group* is a group name, *last* and *first* are the last and first article numbers, and *flag* usually takes one of these values:

- y: Local postings and articles from peers are allowed.
- m: O grupo é moderado e todas as postagens devem ser aprovadas.
- n: No local postings are allowed, only articles from peers.
- j: Articles from peers are filed in the junk group instead.
- x: No local postings, and articles from peers are ignored.
- =foo.bar: Articles are filed in the foo.bar group instead.

If *flag* has another value, then the status of the newsgroup should be considered unknown.

This command can return very large results, especially if *group_pattern* is not specified. It is best to cache the results offline unless you really need to refresh them.

Alterado na versão 3.2: *group_pattern* foi adicionado.

NNTP.**descriptions**(*grouppattern*)
Send a `LIST NEWSGROUPS` command, where *grouppattern* is a wildmat string as specified in **RFC 3977** (it's essentially the same as DOS or UNIX shell wildcard strings). Return a pair `(response, descriptions)`, where *descriptions* is a dictionary mapping group names to textual descriptions.

```
>>> resp, descs = s.descriptions('gmane.comp.python.*')
>>> len(descs)
295
>>> descs.popitem()
('gmane.comp.python.bio.general', 'BioPython discussion list (Moderated)')
```

NNTP.**description**(*group*)
Get a description for a single group *group*. If more than one group matches (if 'group' is a real wildmat string), return the first match. If no group matches, return an empty string.

This elides the response code from the server. If the response code is needed, use *descriptions()*.

NNTP.**group**(*name*)
Send a `GROUP` command, where *name* is the group name. The group is selected as the current group, if it exists. Return a tuple `(response, count, first, last, name)` where *count* is the (estimated) number of articles in the group, *first* is the first article number in the group, *last* is the last article number in the group, and *name* is the group name.

NNTP.**over**(*message_spec*, *, *file=None*)
Send an `OVER` command, or an `XOVER` command on legacy servers. *message_spec* can be either a string representing a message id, or a `(first, last)` tuple of numbers indicating a range of articles in the current group, or a `(first, None)` tuple indicating a range of articles starting from *first* to the last article in the current group, or *None* to select the current article in the current group.

Return a pair `(response, overviews)`. *overviews* is a list of `(article_number, overview)` tuples, one for each article selected by *message_spec*. Each *overview* is a dictionary with the same number of items, but this number depends on the server. These items are either message headers (the key is then the lower-cased header name) or metadata items (the key is then the metadata name prepended with `":"`). The following items are guaranteed to be present by the NNTP specification:

- the `subject`, `from`, `date`, `message-id` and `references` headers
- the `:bytes` metadata: the number of bytes in the entire raw article (including headers and body)
- the `:lines` metadata: the number of lines in the article body

The value of each item is either a string, or *None* if not present.

It is advisable to use the *decode_header()* function on header values when they may contain non-ASCII characters:

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, overviews = s.over((last, last))
>>> art_num, over = overviews[0]
>>> art_num
117216
>>> list(over.keys())
['xref', 'from', ':lines', ':bytes', 'references', 'date', 'message-id', 'subject
↪']
>>> over['from']
'=?UTF-8?B?Ik1hcnRpbiB2LiBMw7Z3aXMi?= <martin@v.loewis.de>'
>>> nntplib.decode_header(over['from'])
'"Martin v. Löwis" <martin@v.loewis.de>'
```

Novo na versão 3.2.

NNTP.**help**(*, *file=None*)
Send a HELP command. Return a pair `(response, list)` where *list* is a list of help strings.

NNTP.**stat**(*message_spec=None*)
Send a STAT command, where *message_spec* is either a message id (enclosed in '<' and '>') or an article number in the current group. If *message_spec* is omitted or *None*, the current article in the current group is considered. Return a triple `(response, number, id)` where *number* is the article number and *id* is the message id.

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, number, message_id = s.stat(first)
>>> number, message_id
(9099, '<20030112190404.GE29873@epoch.metaslash.com>')
```

NNTP.**next**()
Send a NEXT command. Return as for *stat()*.

NNTP.**last**()
Send a LAST command. Return as for *stat()*.

NNTP.**article**(*message_spec=None*, *, *file=None*)
Send an ARTICLE command, where *message_spec* has the same meaning as for *stat()*. Return a tuple `(response, info)` where *info* is a *namedtuple* with three attributes *number*, *message_id* and *lines* (in that order). *number* is the article number in the group (or 0 if the information is not available), *message_id* the message id as a string, and *lines* a list of lines (without terminating newlines) comprising the raw message including headers and body.

```
>>> resp, info = s.article('<20030112190404.GE29873@epoch.metaslash.com>')
>>> info.number
0
>>> info.message_id
'<20030112190404.GE29873@epoch.metaslash.com>'
>>> len(info.lines)
65
>>> info.lines[0]
b'Path: main.gmane.org!not-for-mail'
>>> info.lines[1]
b'From: Neal Norwitz <neal@metaslash.com>'
>>> info.lines[-3:]
[b'There is a patch for 2.3 as well as 2.2.', b'', b'Neal']
```

NNTP.**head**(*message_spec=None*, *, *file=None*)
Same as *article()*, but sends a HEAD command. The *lines* returned (or written to *file*) will only contain the message headers, not the body.

NNTP.**body**(*message_spec=None*, *, *file=None*)
Same as *article()*, but sends a BODY command. The *lines* returned (or written to *file*) will only contain the message body, not the headers.

NNTP.**post**(*data*)
Post an article using the POST command. The *data* argument is either a *file object* opened for binary reading, or any iterable of bytes objects (representing raw lines of the article to be posted). It should represent a well-formed news article, including the required headers. The *post()* method automatically escapes lines beginning with . and appends the termination line.

If the method succeeds, the server's response is returned. If the server refuses posting, a *NNTPReplyError* is raised.

NNTP.**ihave**(*message_id*, *data*)
Send an IHAVE command. *message_id* is the id of the message to send to the server (enclosed in '<' and '>'). The *data* parameter and the return value are the same as for *post()*.

NNTP.**date**()
Return a pair (response, date). *date* is a *datetime* object containing the current date and time of the server.

NNTP.**slave**()
Send a SLAVE command. Return the server's *response*.

NNTP.**set_debuglevel**(*level*)
Set the instance's debugging level. This controls the amount of debugging output printed. The default, 0, produces no debugging output. A value of 1 produces a moderate amount of debugging output, generally a single line per request or response. A value of 2 or higher produces the maximum amount of debugging output, logging each line sent and received on the connection (including message text).

The following are optional NNTP extensions defined in **RFC 2980**. Some of them have been superseded by newer commands in **RFC 3977**.

NNTP.**xhdr**(*hdr*, *str*, *, *file=None*)
Send an XHDR command. The *hdr* argument is a header keyword, e.g. 'subject'. The *str* argument should have the form 'first-last' where *first* and *last* are the first and last article numbers to search. Return a pair (response, list), where *list* is a list of pairs (id, text), where *id* is an article number (as a string) and *text* is the text of the requested header for that article. If the *file* parameter is supplied, then the output of the XHDR command is stored in a file. If *file* is a string, then the method will open a file with that name, write to it then close it. If *file* is a *file object*, then it will start calling write() on it to store the lines of the command output. If *file* is supplied, then the returned *list* is an empty list.

NNTP.**xover**(*start*, *end*, *, *file=None*)
Send an XOVER command. *start* and *end* are article numbers delimiting the range of articles to select. The return value is the same of for *over()*. It is recommended to use *over()* instead, since it will automatically use the newer OVER command if available.

NNTP.**xpath**(*id*)
Return a pair (resp, path), where *path* is the directory path to the article with message ID *id*. Most of the time, this extension is not enabled by NNTP server administrators.

Obsoleto desde a versão 3.3: The XPATH extension is not actively used.

22.16.2 Funções utilitárias

The module also defines the following utility function:

nntplib.**decode_header**(*header_str*)
Decode a header value, un-escaping any escaped non-ASCII characters. *header_str* must be a *str* object. The unescaped value is returned. Using this function is recommended to display some headers in a human readable form:

```
>>> decode_header("Some subject")
'Some subject'
>>> decode_header("=?ISO-8859-15?Q?D=E9buter_en_Python?=")
'Débuter en Python'
>>> decode_header("Re: =?UTF-8?B?cHJvYmzDqG1lIGRlIG1hdHJpY2U=?=")
'Re: problème de matrice'
```

22.17 smtplib — SMTP protocol client

Código Fonte: Lib/smtplib.py

The *smtplib* module defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon. For details of SMTP and ESMTP operation, consult **RFC 821** (Simple Mail Transfer Protocol) and **RFC 1869** (SMTP Service Extensions).

class smtplib.**SMTP**(*host=''*, *port=0*, *local_hostname=None*[, *timeout*], *source_address=None*)

An *SMTP* instance encapsulates an SMTP connection. It has methods that support a full repertoire of SMTP and ESMTP operations. If the optional host and port parameters are given, the SMTP *connect()* method is called with those parameters during initialization. If specified, *local_hostname* is used as the FQDN of the local host in the HELO/EHLO command. Otherwise, the local hostname is found using *socket.getfqdn()*. If the *connect()* call returns anything other than a success code, an *SMTPConnectError* is raised. The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). If the timeout expires, *socket.timeout* is raised. The optional source_address parameter allows binding to some specific source address in a machine with multiple network interfaces, and/or to some specific source TCP port. It takes a 2-tuple (host, port), for the socket to bind to as its source address before connecting. If omitted (or if host or port are `''` and/or 0 respectively) the OS default behavior will be used.

For normal use, you should only require the initialization/connect, *sendmail()*, and *SMTP.quit()* methods. An example is included below.

The *SMTP* class supports the `with` statement. When used like this, the SMTP `QUIT` command is issued automatically when the `with` statement exits. E.g.:

```
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
...
(250, b'Ok')
>>>
```

Alterado na versão 3.3: Suporte para a instrução `with` foi adicionado.

Alterado na versão 3.3: source_address argument was added.

Novo na versão 3.5: The SMTPUTF8 extension (**RFC 6531**) is now supported.

class smtplib.**SMTP_SSL**(*host=''*, *port=0*, *local_hostname=None*, *keyfile=None*, *certfile=None*[, *timeout*], *context=None*, *source_address=None*)

An *SMTP_SSL* instance behaves exactly the same as instances of *SMTP*. *SMTP_SSL* should be used for situations where SSL is required from the beginning of the connection and using `starttls()` is not appropriate. If *host* is not specified, the local host is used. If *port* is zero, the standard SMTP-over-SSL port (465) is used. The optional arguments *local_hostname*, *timeout* and *source_address* have the same meaning as they do in the *SMTP* class. *context*, also optional, can contain a *SSLContext* and allows configuring various aspects of the secure connection. Please read *Considerações de segurança* for best practices.

keyfile and *certfile* are a legacy alternative to *context*, and can point to a PEM formatted private key and certificate chain file for the SSL connection.

Alterado na versão 3.3: *context* foi adicionado.

Alterado na versão 3.3: source_address argument was added.

Alterado na versão 3.4: The class now supports hostname check with *ssl.SSLContext.check_hostname* and *Server Name Indication* (see *ssl.HAS_SNI*).

Obsoleto desde a versão 3.6: *keyfile* and *certfile* are deprecated in favor of *context*. Please use `ssl.SSLContext.load_cert_chain()` instead, or let `ssl.create_default_context()` select the system's trusted CA certificates for you.

class smtplib.**LMTP**(*host=''*, *port=LMTP_PORT*, *local_hostname=None*, *source_address=None*)

The LMTP protocol, which is very similar to ESMTP, is heavily based on the standard SMTP client. It's common to use Unix sockets for LMTP, so our `connect()` method must support that as well as a regular host:port server. The optional arguments local_hostname and source_address have the same meaning as they do in the `SMTP` class. To specify a Unix socket, you must use an absolute path for *host*, starting with a '/'.

Authentication is supported, using the regular SMTP mechanism. When using a Unix socket, LMTP generally don't support or require any authentication, but your mileage might vary.

A nice selection of exceptions is defined as well:

exception smtplib.**SMTPException**

Subclass of `OSError` that is the base exception class for all the other exceptions provided by this module.

Alterado na versão 3.4: SMTPException became subclass of `OSError`

exception smtplib.**SMTPServerDisconnected**

This exception is raised when the server unexpectedly disconnects, or when an attempt is made to use the `SMTP` instance before connecting it to a server.

exception smtplib.**SMTPResponseException**

Base class for all exceptions that include an SMTP error code. These exceptions are generated in some instances when the SMTP server returns an error code. The error code is stored in the `smtp_code` attribute of the error, and the `smtp_error` attribute is set to the error message.

exception smtplib.**SMTPSenderRefused**

Sender address refused. In addition to the attributes set by on all `SMTPResponseException` exceptions, this sets 'sender' to the string that the SMTP server refused.

exception smtplib.**SMTPRecipientsRefused**

All recipient addresses refused. The errors for each recipient are accessible through the attribute `recipients`, which is a dictionary of exactly the same sort as `SMTP.sendmail()` returns.

exception smtplib.**SMTPDataError**

The SMTP server refused to accept the message data.

exception smtplib.**SMTPConnectError**

Error occurred during establishment of a connection with the server.

exception smtplib.**SMTPHeloError**

The server refused our `HELO` message.

exception smtplib.**SMTPNotSupportedError**

The command or option attempted is not supported by the server.

Novo na versão 3.5.

exception smtplib.**SMTPAuthenticationError**

SMTP authentication went wrong. Most probably the server didn't accept the username/password combination provided.

Ver também:

RFC 821 - Simple Mail Transfer Protocol Protocol definition for SMTP. This document covers the model, operating procedure, and protocol details for SMTP.

RFC 1869 - SMTP Service Extensions Definition of the ESMTP extensions for SMTP. This describes a framework for extending SMTP with new commands, supporting dynamic discovery of the commands provided by the server, and defines a few additional commands.

22.17.1 Objetos SMTP

An *SMTP* instance has the following methods:

SMTP.**set_debuglevel**(*level*)
Set the debug output level. A value of 1 or `True` for *level* results in debug messages for connection and for all messages sent to and received from the server. A value of 2 for *level* results in these messages being timestamped.

Alterado na versão 3.5: Adicionado o nível de depuração 2.

SMTP.**docmd**(*cmd*, *args=''*)
Send a command *cmd* to the server. The optional argument *args* is simply concatenated to the command, separated by a space.

This returns a 2-tuple composed of a numeric response code and the actual response line (multiline responses are joined into one long line.)

In normal operation it should not be necessary to call this method explicitly. It is used to implement other methods and may be useful for testing private extensions.

If the connection to the server is lost while waiting for the reply, *SMTPServerDisconnected* will be raised.

SMTP.**connect**(*host='localhost'*, *port=0*)
Connect to a host on a given port. The defaults are to connect to the local host at the standard SMTP port (25). If the hostname ends with a colon (`':'`) followed by a number, that suffix will be stripped off and the number interpreted as the port number to use. This method is automatically invoked by the constructor if a host is specified during instantiation. Returns a 2-tuple of the response code and message sent by the server in its connection response.

SMTP.**helo**(*name=''*)
Identify yourself to the SMTP server using `HELO`. The hostname argument defaults to the fully qualified domain name of the local host. The message returned by the server is stored as the `helo_resp` attribute of the object.

In normal operation it should not be necessary to call this method explicitly. It will be implicitly called by the *sendmail()* when necessary.

SMTP.**ehlo**(*name=''*)
Identify yourself to an ESMTP server using `EHLO`. The hostname argument defaults to the fully qualified domain name of the local host. Examine the response for ESMTP option and store them for use by *has_extn()*. Also sets several informational attributes: the message returned by the server is stored as the `ehlo_resp` attribute, `does_esmtp` is set to true or false depending on whether the server supports ESMTP, and `esmtp_features` will be a dictionary containing the names of the SMTP service extensions this server supports, and their parameters (if any).

Unless you wish to use *has_extn()* before sending mail, it should not be necessary to call this method explicitly. It will be implicitly called by *sendmail()* when necessary.

SMTP.**ehlo_or_helo_if_needed**()
This method calls *ehlo()* and/or *helo()* if there has been no previous `EHLO` or `HELO` command this session. It tries ESMTP `EHLO` first.

SMTPHeloError The server didn't reply properly to the `HELO` greeting.

SMTP.**has_extn**(*name*)
Return *True* if *name* is in the set of SMTP service extensions returned by the server, *False* otherwise. Case is ignored.

SMTP.**verify**(*address*)
Check the validity of an address on this server using SMTP `VRFY`. Returns a tuple consisting of code 250 and a full **RFC 822** address (including human name) if the user address is valid. Otherwise returns an SMTP error code of 400 or greater and an error string.

Nota: Many sites disable SMTP `VRFY` in order to foil spammers.

SMTP.**login**(*user*, *password*, *, *initial_response_ok=True*)

Log in on an SMTP server that requires authentication. The arguments are the username and the password to authenticate with. If there has been no previous `EHLO` or `HELO` command this session, this method tries ESMTP `EHLO` first. This method will return normally if the authentication was successful, or may raise the following exceptions:

SMTPHeloError The server didn't reply properly to the `HELO` greeting.

SMTPAuthenticationError The server didn't accept the username/password combination.

SMTPNotSupportedError The `AUTH` command is not supported by the server.

SMTPException No suitable authentication method was found.

Each of the authentication methods supported by *smtplib* are tried in turn if they are advertised as supported by the server. See *auth()* for a list of supported authentication methods. *initial_response_ok* is passed through to *auth()*.

Optional keyword argument *initial_response_ok* specifies whether, for authentication methods that support it, an "initial response" as specified in **RFC 4954** can be sent along with the `AUTH` command, rather than requiring a challenge/response.

Alterado na versão 3.5: *SMTPNotSupportedError* may be raised, and the *initial_response_ok* parameter was added.

SMTP.**auth**(*mechanism*, *authobject*, *, *initial_response_ok=True*)

Issue an `SMTP AUTH` command for the specified authentication *mechanism*, and handle the challenge response via *authobject*.

mechanism specifies which authentication mechanism is to be used as argument to the `AUTH` command; the valid values are those listed in the `auth` element of `esmtp_features`.

authobject must be a callable object taking an optional single argument:

```
data = authobject(challenge=None)
```

If optional keyword argument *initial_response_ok* is true, `authobject()` will be called first with no argument. It can return the **RFC 4954** "initial response" ASCII `str` which will be encoded and sent with the `AUTH` command as below. If the `authobject()` does not support an initial response (e.g. because it requires a challenge), it should return `None` when called with `challenge=None`. If *initial_response_ok* is false, then `authobject()` will not be called first with `None`.

If the initial response check returns `None`, or if *initial_response_ok* is false, `authobject()` will be called to process the server's challenge response; the *challenge* argument it is passed will be a `bytes`. It should return ASCII `str` *data* that will be base64 encoded and sent to the server.

The `SMTP` class provides `authobjects` for the `CRAM-MD5`, `PLAIN`, and `LOGIN` mechanisms; they are named `SMTP.auth_cram_md5`, `SMTP.auth_plain`, and `SMTP.auth_login` respectively. They all require that the `user` and `password` properties of the `SMTP` instance are set to appropriate values.

User code does not normally need to call `auth` directly, but can instead call the *login()* method, which will try each of the above mechanisms in turn, in the order listed. `auth` is exposed to facilitate the implementation of authentication methods not (or not yet) supported directly by *smtplib*.

Novo na versão 3.5.

SMTP.**starttls**(*keyfile=None*, *certfile=None*, *context=None*)

Put the SMTP connection in TLS (Transport Layer Security) mode. All SMTP commands that follow will be encrypted. You should then call *ehlo()* again.

If *keyfile* and *certfile* are provided, they are used to create an *ssl.SSLContext*.

Optional *context* parameter is an *ssl.SSLContext* object; This is an alternative to using a keyfile and a certfile and if specified both *keyfile* and *certfile* should be None.

If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first.

Obsoleto desde a versão 3.6: *keyfile* and *certfile* are deprecated in favor of *context*. Please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system's trusted CA certificates for you.

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTPNotSupportedError The server does not support the STARTTLS extension.

RuntimeError SSL/TLS support is not available to your Python interpreter.

Alterado na versão 3.3: *context* foi adicionado.

Alterado na versão 3.4: The method now supports hostname check with SSLContext.check_hostname and *Server Name Indicator* (see *HAS_SNI*).

Alterado na versão 3.5: The error raised for lack of STARTTLS support is now the *SMTPNotSupportedError* subclass instead of the base *SMTPException*.

SMTP.**sendmail**(*from_addr*, *to_addrs*, *msg*, *mail_options=()*, *rcpt_options=()*)

Send mail. The required arguments are an **RFC 822** from-address string, a list of **RFC 822** to-address strings (a bare string will be treated as a list with 1 address), and a message string. The caller may pass a list of ESMTP options (such as 8bitmime) to be used in MAIL FROM commands as *mail_options*. ESMTP options (such as DSN commands) that should be used with all RCPT commands can be passed as *rcpt_options*. (If you need to use different ESMTP options to different recipients you have to use the low-level methods such as mail(), rcpt() and data() to send the message.)

Nota: The *from_addr* and *to_addrs* parameters are used to construct the message envelope used by the transport agents. sendmail does not modify the message headers in any way.

---

msg may be a string containing characters in the ASCII range, or a byte string. A string is encoded to bytes using the ascii codec, and lone \r and \n characters are converted to \r\n characters. A byte string is not modified.

If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first. If the server does ESMTP, message size and each of the specified options will be passed to it (if the option is in the feature set the server advertises). If EHLO fails, HELO will be tried and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one recipient. Otherwise it will raise an exception. That is, if this method does not raise an exception, then someone should get your mail. If this method does not raise an exception, it returns a dictionary, with one entry for each recipient that was refused. Each entry contains a tuple of the SMTP error code and the accompanying error message sent by the server.

If SMTPUTF8 is included in *mail_options*, and the server supports it, *from_addr* and *to_addrs* may contain non-ASCII characters.

This method may raise the following exceptions:

SMTPRecipientsRefused All recipients were refused. Nobody got the mail. The recipients attribute of the exception object is a dictionary with information about the refused recipients (like the one returned when at least one recipient was accepted).

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTPSenderRefused The server didn't accept the *from_addr*.

SMTPDataError The server replied with an unexpected error code (other than a refusal of a recipient).

SMTPNotSupportedError SMTPUTF8 was given in the *mail_options* but is not supported by the server.

Unless otherwise noted, the connection will be open even after an exception is raised.

Alterado na versão 3.2: *msg* may be a byte string.

Alterado na versão 3.5: SMTPUTF8 support added, and *SMTPNotSupportedError* may be raised if SMTPUTF8 is specified but the server does not support it.

SMTP.**send_message**(*msg*, *from_addr=None*, *to_addrs=None*, *mail_options=()*, *rcpt_options=()*)

This is a convenience method for calling *sendmail()* with the message represented by an *email.message.Message* object. The arguments have the same meaning as for *sendmail()*, except that *msg* is a Message object.

If *from_addr* is None or *to_addrs* is None, send_message fills those arguments with addresses extracted from the headers of *msg* as specified in **RFC 5322**: *from_addr* is set to the *Sender* field if it is present, and otherwise to the *From* field. *to_addrs* combines the values (if any) of the *To*, *Cc*, and *Bcc* fields from *msg*. If exactly one set of *Resent-** headers appear in the message, the regular headers are ignored and the *Resent-** headers are used instead. If the message contains more than one set of *Resent-** headers, a *ValueError* is raised, since there is no way to unambiguously detect the most recent set of *Resent-* headers.

send_message serializes *msg* using *BytesGenerator* with \r\n as the *linesep*, and calls *sendmail()* to transmit the resulting message. Regardless of the values of *from_addr* and *to_addrs*, send_message does not transmit any *Bcc* or *Resent-Bcc* headers that may appear in *msg*. If any of the addresses in *from_addr* and *to_addrs* contain non-ASCII characters and the server does not advertise SMTPUTF8 support, an SMTPNotSupported error is raised. Otherwise the Message is serialized with a clone of its *policy* with the *utf8* attribute set to True, and SMTPUTF8 and BODY=8BITMIME are added to *mail_options*.

Novo na versão 3.2.

Novo na versão 3.5: Support for internationalized addresses (SMTPUTF8).

SMTP.**quit**()

Terminate the SMTP session and close the connection. Return the result of the SMTP QUIT command.

Low-level methods corresponding to the standard SMTP/ESMTP commands HELP, RSET, NOOP, MAIL, RCPT, and DATA are also supported. Normally these do not need to be called directly, so they are not documented here. For details, consult the module code.

22.17.2 Exemplo SMTP

This example prompts the user for addresses needed in the message envelope ('To' and 'From' addresses), and the message to be delivered. Note that the headers to be included with the message must be included in the message as entered; this example doesn't do any processing of the **RFC 822** headers. In particular, the 'To' and 'From' addresses must be included in the message headers explicitly.

```
import smtplib

def prompt(prompt):
    return input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs  = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\n"
       % (fromaddr, ", ".join(toaddrs)))
```

(continua na próxima página)

(continuação da página anterior)

```
while True:
    try:
        line = input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print("Message length is", len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

Nota: In general, you will want to use the `email` package's features to construct an email message, which you can then send via `send_message()`; see *email: Exemplos*.

22.18 `smtpd` — Serviços SMTP

Código Fonte: Lib/smtpd.py

This module offers several classes to implement SMTP (email) servers.

Ver também:

The aiosmtpd package is a recommended replacement for this module. It is based on `asyncio` and provides a more straightforward API. `smtpd` should be considered deprecated.

Several server implementations are present; one is a generic do-nothing implementation, which can be overridden, while the other two offer specific mail-sending strategies.

Additionally the SMTPChannel may be extended to implement very specific interaction behaviour with SMTP clients.

The code supports **RFC 5321**, plus the **RFC 1870** SIZE and **RFC 6531** SMTPUTF8 extensions.

22.18.1 Objetos SMTPServer

class smtpd.**SMTPServer**(*localaddr*, *remoteaddr*, *data_size_limit=33554432*, *map=None*, *enable_SMTPUTF8=False*, *decode_data=False*)

Create a new `SMTPServer` object, which binds to local address *localaddr*. It will treat *remoteaddr* as an upstream SMTP relayer. Both *localaddr* and *remoteaddr* should be a *(host, port)* tuple. The object inherits from `asyncore.dispatcher`, and so will insert itself into `asyncore`'s event loop on instantiation.

data_size_limit specifies the maximum number of bytes that will be accepted in a `DATA` command. A value of `None` or `0` means no limit.

map is the socket map to use for connections (an initially empty dictionary is a suitable value). If not specified the `asyncore` global socket map is used.

enable_SMTPUTF8 determines whether the `SMTPUTF8` extension (as defined in **RFC 6531**) should be enabled. The default is `False`. When `True`, `SMTPUTF8` is accepted as a parameter to the `MAIL` command and when

present is passed to *process_message()* in the `kwargs['mail_options']` list. *decode_data* and *enable_SMTPUTF8* cannot be set to `True` at the same time.

decode_data specifies whether the data portion of the SMTP transaction should be decoded using UTF-8. When *decode_data* is `False` (the default), the server advertises the `8BITMIME` extension (**RFC 6152**), accepts the `BODY=8BITMIME` parameter to the `MAIL` command, and when present passes it to *process_message()* in the `kwargs['mail_options']` list. *decode_data* and *enable_SMTPUTF8* cannot be set to `True` at the same time.

process_message(*peer*, *mailfrom*, *rcpttos*, *data*, ***kwargs*)
Raise a *NotImplementedError* exception. Override this in subclasses to do something useful with this message. Whatever was passed in the constructor as *remoteaddr* will be available as the `_remoteaddr` attribute. *peer* is the remote host's address, *mailfrom* is the envelope originator, *rcpttos* are the envelope recipients and *data* is a string containing the contents of the e-mail (which should be in **RFC 5321** format).

If the *decode_data* constructor keyword is set to `True`, the *data* argument will be a unicode string. If it is set to `False`, it will be a bytes object.

kwargs is a dictionary containing additional information. It is empty if `decode_data=True` was given as an init argument, otherwise it contains the following keys:

***mail_options*:** a list of all received parameters to the `MAIL` command (the elements are uppercase strings; example: `['BODY=8BITMIME', 'SMTPUTF8']`).

***rcpt_options*:** same as *mail_options* but for the `RCPT` command. Currently no `RCPT TO` options are supported, so for now this will always be an empty list.

Implementations of `process_message` should use the `**kwargs` signature to accept arbitrary keyword arguments, since future feature enhancements may add keys to the kwargs dictionary.

Return `None` to request a normal `250 Ok` response; otherwise return the desired response string in **RFC 5321** format.

channel_class
Override this in subclasses to use a custom *SMTPChannel* for managing SMTP clients.

Novo na versão 3.4: The *map* constructor argument.

Alterado na versão 3.5: *localaddr* and *remoteaddr* may now contain IPv6 addresses.

Novo na versão 3.5: The *decode_data* and *enable_SMTPUTF8* constructor parameters, and the *kwargs* parameter to *process_message()* when *decode_data* is `False`.

Alterado na versão 3.6: *decode_data* is now `False` by default.

22.18.2 DebuggingServer Objects

class smtpd.**DebuggingServer**(*localaddr*, *remoteaddr*)
Create a new debugging server. Arguments are as per *SMTPServer*. Messages will be discarded, and printed on stdout.

22.18.3 Objetos PureProxy

class smtpd.**PureProxy**(*localaddr*, *remoteaddr*)
Create a new pure proxy server. Arguments are as per *SMTPServer*. Everything will be relayed to *remoteaddr*. Note that running this has a good chance to make you into an open relay, so please be careful.

22.18.4 MailmanProxy Objects

class smtpd.**MailmanProxy**(*localaddr*, *remoteaddr*)
Create a new pure proxy server. Arguments are as per *SMTPServer*. Everything will be relayed to *remoteaddr*, unless local mailman configurations knows about an address, in which case it will be handled via mailman. Note that running this has a good chance to make you into an open relay, so please be careful.

22.18.5 SMTPChannel Objects

class smtpd.**SMTPChannel**(*server*, *conn*, *addr*, *data_size_limit=33554432*, *map=None*, *enable_SMTPUTF8=False*, *decode_data=False*)
Create a new *SMTPChannel* object which manages the communication between the server and a single SMTP client.

conn and *addr* are as per the instance variables described below.

data_size_limit specifies the maximum number of bytes that will be accepted in a DATA command. A value of None or 0 means no limit.

enable_SMTPUTF8 determines whether the SMTPUTF8 extension (as defined in **RFC 6531**) should be enabled. The default is False. *decode_data* and *enable_SMTPUTF8* cannot be set to True at the same time.

A dictionary can be specified in *map* to avoid using a global socket map.

decode_data specifies whether the data portion of the SMTP transaction should be decoded using UTF-8. The default is False. *decode_data* and *enable_SMTPUTF8* cannot be set to True at the same time.

To use a custom SMTPChannel implementation you need to override the *SMTPServer.channel_class* of your *SMTPServer*.

Alterado na versão 3.5: The *decode_data* and *enable_SMTPUTF8* parameters were added.

Alterado na versão 3.6: *decode_data* is now False by default.

The *SMTPChannel* has the following instance variables:

smtp_server
Holds the *SMTPServer* that spawned this channel.

conn
Holds the socket object connecting to the client.

addr
Holds the address of the client, the second value returned by *socket.accept*

received_lines
Holds a list of the line strings (decoded using UTF-8) received from the client. The lines have their "\r\n" line ending translated to "\n".

smtp_state
Holds the current state of the channel. This will be either COMMAND initially and then DATA after the client sends a "DATA" line.

seen_greeting
Holds a string containing the greeting sent by the client in its "HELO".

mailfrom
Holds a string containing the address identified in the "MAIL FROM:" line from the client.

rcpttos
Holds a list of strings containing the addresses identified in the "RCPT TO:" lines from the client.

received_data
Holds a string containing all of the data sent by the client during the DATA state, up to but not including the terminating `"\r\n.\r\n"`.

fqdn
Holds the fully-qualified domain name of the server as returned by *socket.getfqdn()*.

peer
Holds the name of the client peer as returned by `conn.getpeername()` where `conn` is *conn*.

The *SMTPChannel* operates by invoking methods named `smtp_<command>` upon reception of a command line from the client. Built into the base *SMTPChannel* class are methods for handling the following commands (and responding to them appropriately):

Comando	Action taken
HELO	Accepts the greeting from the client and stores it in *seen_greeting*. Sets server to base command mode.
EHLO	Accepts the greeting from the client and stores it in *seen_greeting*. Sets server to extended command mode.
NOOP	Takes no action.
QUIT	Closes the connection cleanly.
MAIL	Accepts the "MAIL FROM:" syntax and stores the supplied address as *mailfrom*. In extended command mode, accepts the **RFC 1870** SIZE attribute and responds appropriately based on the value of *data_size_limit*.
RCPT	Accepts the "RCPT TO:" syntax and stores the supplied addresses in the *rcpttos* list.
RSET	Resets the *mailfrom*, *rcpttos*, and *received_data*, but not the greeting.
DATA	Sets the internal state to `DATA` and stores remaining lines from the client in *received_data* until the terminator `"\r\n.\r\n"` is received.
HELP	Returns minimal information on command syntax
VRFY	Returns code 252 (the server doesn't know if the address is valid)
EXPN	Reports that the command is not implemented.

22.19 telnetlib — cliente Telnet

Código Fonte: Lib/telnetlib.py

The *telnetlib* module provides a *Telnet* class that implements the Telnet protocol. See **RFC 854** for details about the protocol. In addition, it provides symbolic constants for the protocol characters (see below), and for the telnet options. The symbolic names of the telnet options follow the definitions in `arpa/telnet.h`, with the leading `TELOPT_` removed. For symbolic names of options which are traditionally not included in `arpa/telnet.h`, see the module source itself.

The symbolic constants for the telnet commands are: IAC, DONT, DO, WONT, WILL, SE (Subnegotiation End), NOP (No Operation), DM (Data Mark), BRK (Break), IP (Interrupt process), AO (Abort output), AYT (Are You There), EC (Erase Character), EL (Erase Line), GA (Go Ahead), SB (Subnegotiation Begin).

class telnetlib.**Telnet**(*host=None*, *port=0*[, *timeout*])

Telnet represents a connection to a Telnet server. The instance is initially not connected by default; the *open()* method must be used to establish a connection. Alternatively, the host name and optional port number can be passed to the constructor too, in which case the connection to the server will be established before the constructor returns. The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).

Do not reopen an already connected instance.

This class has many `read_*()` methods. Note that some of them raise *EOFError* when the end of the connection is read, because they can return an empty string for other reasons. See the individual descriptions below.

Um objeto *Telnet* é um gerenciador de contexto e pode ser usado em uma instrução `with`. Quando o bloco `with` termina, o método *close()* é chamado:

```
>>> from telnetlib import Telnet
>>> with Telnet('localhost', 23) as tn:
...     tn.interact()
...
```

Alterado na versão 3.6: Adicionado suporte a gerenciador de contexto

Ver também:

RFC 854 - Especificação do Protocolo Telnet Definition of the Telnet protocol.

22.19.1 Objetos Telnet

Telnet instances have the following methods:

Telnet.**read_until**(*expected*, *timeout=None*)

Read until a given byte string, *expected*, is encountered or until *timeout* seconds have passed.

When no match is found, return whatever is available instead, possibly empty bytes. Raise *EOFError* if the connection is closed and no cooked data is available.

Telnet.**read_all**()

Read all data until EOF as bytes; block until connection closed.

Telnet.**read_some**()

Read at least one byte of cooked data unless EOF is hit. Return `b''` if EOF is hit. Block if no data is immediately available.

Telnet.**read_very_eager**()

Read everything that can be without blocking in I/O (eager).

Raise *EOFError* if connection closed and no cooked data available. Return `b''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

Telnet.**read_eager**()

Read readily available data.

Raise *EOFError* if connection closed and no cooked data available. Return `b''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

Telnet.**read_lazy**()

Process and return data already in the queues (lazy).

Raise *EOFError* if connection closed and no data available. Return `b''` if no cooked data available otherwise. Do not block unless in the midst of an IAC sequence.

`Telnet.`**`read_very_lazy`**`()`
Return any data available in the cooked queue (very lazy).

Raise *EOFError* if connection closed and no data available. Return `b''` if no cooked data available otherwise. This method never blocks.

`Telnet.`**`read_sb_data`**`()`
Return the data collected between a SB/SE pair (suboption begin/end). The callback should access these data when it was invoked with a `SE` command. This method never blocks.

`Telnet.`**`open`**`(`*host*, *port=0*[, *timeout*]`)`
Connect to a host. The optional second argument is the port number, which defaults to the standard Telnet port (23). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).

Do not try to reopen an already connected instance.

`Telnet.`**`msg`**`(`*msg*, **args*`)`
Print a debug message when the debug level is > 0. If extra arguments are present, they are substituted in the message using the standard string formatting operator.

`Telnet.`**`set_debuglevel`**`(`*debuglevel*`)`
Set the debug level. The higher the value of *debuglevel*, the more debug output you get (on `sys.stdout`).

`Telnet.`**`close`**`()`
Close the connection.

`Telnet.`**`get_socket`**`()`
Return the socket object used internally.

`Telnet.`**`fileno`**`()`
Return the file descriptor of the socket object used internally.

`Telnet.`**`write`**`(`*buffer*`)`
Write a byte string to the socket, doubling any IAC characters. This can block if the connection is blocked. May raise *OSError* if the connection is closed.

Alterado na versão 3.3: This method used to raise *socket.error*, which is now an alias of *OSError*.

`Telnet.`**`interact`**`()`
Interaction function, emulates a very dumb Telnet client.

`Telnet.`**`mt_interact`**`()`
Multithreaded version of *interact()*.

`Telnet.`**`expect`**`(`*list*, *timeout=None*`)`
Read until one from a list of a regular expressions matches.

The first argument is a list of regular expressions, either compiled (*regex objects*) or uncompiled (byte strings). The optional second argument is a timeout, in seconds; the default is to block indefinitely.

Return a tuple of three items: the index in the list of the first regular expression that matches; the match object returned; and the bytes read up till and including the match.

If end of file is found and no bytes were read, raise *EOFError*. Otherwise, when nothing matches, return `(-1, None, data)` where *data* is the bytes received so far (may be empty bytes if a timeout happened).

If a regular expression ends with a greedy match (such as `.*`) or if more than one expression can match the same input, the results are non-deterministic, and may depend on the I/O timing.

Telnet.**set_option_negotiation_callback**(*callback*)
Each time a telnet option is read on the input flow, this *callback* (if set) is called with the following parameters: callback(telnet socket, command (DO/DONT/WILL/WONT), option). No other action is done afterwards by telnetlib.

22.19.2 Telnet Example

A simple example illustrating typical use:

```
import getpass
import telnetlib

HOST = "localhost"
user = input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.read_until(b"login: ")
tn.write(user.encode('ascii') + b"\n")
if password:
    tn.read_until(b"Password: ")
    tn.write(password.encode('ascii') + b"\n")

tn.write(b"ls\n")
tn.write(b"exit\n")

print(tn.read_all().decode('ascii'))
```

22.20 `uuid` — UUID objects according to RFC 4122

Código Fonte: Lib/uuid.py

This module provides immutable *UUID* objects (the *UUID* class) and the functions *uuid1()*, *uuid3()*, *uuid4()*, *uuid5()* for generating version 1, 3, 4, and 5 UUIDs as specified in **RFC 4122**.

If all you want is a unique ID, you should probably call *uuid1()* or *uuid4()*. Note that *uuid1()* may compromise privacy since it creates a UUID containing the computer's network address. *uuid4()* creates a random UUID.

Depending on support from the underlying platform, *uuid1()* may or may not return a "safe" UUID. A safe UUID is one which is generated using synchronization methods that ensure no two processes can obtain the same UUID. All instances of *UUID* have an `is_safe` attribute which relays any information about the UUID's safety, using this enumeration:

class uuid.**SafeUUID**
Novo na versão 3.7.

safe
The UUID was generated by the platform in a multiprocessing-safe way.

unsafe
The UUID was not generated in a multiprocessing-safe way.

unknown
The platform does not provide information on whether the UUID was generated safely or not.

class uuid.**UUID**(*hex=None*, *bytes=None*, *bytes_le=None*, *fields=None*, *int=None*, *version=None*, ***, *is_safe=SafeUUID.unknown*)

Create a UUID from either a string of 32 hexadecimal digits, a string of 16 bytes in big-endian order as the *bytes* argument, a string of 16 bytes in little-endian order as the *bytes_le* argument, a tuple of six integers (32-bit *time_low*, 16-bit *time_mid*, 16-bit *time_hi_version*, 8-bit *clock_seq_hi_variant*, 8-bit *clock_seq_low*, 48-bit *node*) as the *fields* argument, or a single 128-bit integer as the *int* argument. When a string of hex digits is given, curly braces, hyphens, and a URN prefix are all optional. For example, these expressions all yield the same UUID:

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes=b'\x12\x34\x56\x78'*4)
UUID(bytes_le=b'\x78\x56\x34\x12\x34\x12\x78\x56' +
              b'\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

Exactly one of *hex*, *bytes*, *bytes_le*, *fields*, or *int* must be given. The *version* argument is optional; if given, the resulting UUID will have its variant and version number set according to **RFC 4122**, overriding bits in the given *hex*, *bytes*, *bytes_le*, *fields*, or *int*.

Comparison of UUID objects are made by way of comparing their *UUID.int* attributes. Comparison with a non-UUID object raises a *TypeError*.

`str(uuid)` returns a string in the form `12345678-1234-5678-1234-567812345678` where the 32 hexadecimal digits represent the UUID.

UUID instances have these read-only attributes:

UUID.**bytes**

The UUID as a 16-byte string (containing the six integer fields in big-endian byte order).

UUID.**bytes_le**

The UUID as a 16-byte string (with *time_low*, *time_mid*, and *time_hi_version* in little-endian byte order).

UUID.**fields**

A tuple of the six integer fields of the UUID, which are also available as six individual attributes and two derived attributes:

Campo	Significado
time_low	os primeiros 32 bits do UUID
time_mid	os próximos 16 bits do UUID
time_hi_version	os próximos 16 bits do UUID
clock_seq_hi_variant	os próximos 8 bits do UUID
clock_seq_low	os próximos 8 bits do UUID
node	the last 48 bits of the UUID
time	the 60-bit timestamp
clock_seq	the 14-bit sequence number

UUID.**hex**

The UUID as a 32-character hexadecimal string.

UUID.**int**

The UUID as a 128-bit integer.

UUID.**urn**

The UUID as a URN as specified in **RFC 4122**.

UUID.**variant**
The UUID variant, which determines the internal layout of the UUID. This will be one of the constants *RESERVED_NCS*, *RFC_4122*, *RESERVED_MICROSOFT*, or *RESERVED_FUTURE*.

UUID.**version**
The UUID version number (1 through 5, meaningful only when the variant is *RFC_4122*).

UUID.**is_safe**
An enumeration of *SafeUUID* which indicates whether the platform generated the UUID in a multiprocessing-safe way.

Novo na versão 3.7.

The *uuid* module defines the following functions:

uuid.**getnode**()
Get the hardware address as a 48-bit positive integer. The first time this runs, it may launch a separate program, which could be quite slow. If all attempts to obtain the hardware address fail, we choose a random 48-bit number with the multicast bit (least significant bit of the first octet) set to 1 as recommended in **RFC 4122**. "Hardware address" means the MAC address of a network interface. On a machine with multiple network interfaces, universally administered MAC addresses (i.e. where the second least significant bit of the first octet is *unset*) will be preferred over locally administered MAC addresses, but with no other ordering guarantees.

Alterado na versão 3.7: Universally administered MAC addresses are preferred over locally administered MAC addresses, since the former are guaranteed to be globally unique, while the latter are not.

uuid.**uuid1**(*node=None*, *clock_seq=None*)
Generate a UUID from a host ID, sequence number, and the current time. If *node* is not given, *getnode()* is used to obtain the hardware address. If *clock_seq* is given, it is used as the sequence number; otherwise a random 14-bit sequence number is chosen.

uuid.**uuid3**(*namespace*, *name*)
Generate a UUID based on the MD5 hash of a namespace identifier (which is a UUID) and a name (which is a string).

uuid.**uuid4**()
Generate a random UUID.

uuid.**uuid5**(*namespace*, *name*)
Generate a UUID based on the SHA-1 hash of a namespace identifier (which is a UUID) and a name (which is a string).

The *uuid* module defines the following namespace identifiers for use with *uuid3()* or *uuid5()*.

uuid.**NAMESPACE_DNS**
When this namespace is specified, the *name* string is a fully-qualified domain name.

uuid.**NAMESPACE_URL**
When this namespace is specified, the *name* string is a URL.

uuid.**NAMESPACE_OID**
When this namespace is specified, the *name* string is an ISO OID.

uuid.**NAMESPACE_X500**
When this namespace is specified, the *name* string is an X.500 DN in DER or a text output format.

The *uuid* module defines the following constants for the possible values of the `variant` attribute:

uuid.**RESERVED_NCS**
Reserved for NCS compatibility.

uuid.**RFC_4122**
Specifies the UUID layout given in **RFC 4122**.

uuid.**RESERVED_MICROSOFT**
Reserved for Microsoft compatibility.

uuid.**RESERVED_FUTURE**
Reserved for future definition.

Ver também:

RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace This specification defines a Uniform Resource Name namespace for UUIDs, the internal format of UUIDs, and methods of generating UUIDs.

22.20.1 Exemplo

Here are some examples of typical usage of the *uuid* module:

```
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')

>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```

22.21 socketserver — A framework for network servers

Código Fonte: Lib/socketserver.py

The *socketserver* module simplifies the task of writing network servers.

There are four basic concrete server classes:

class socketserver.**TCPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
This uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. If *bind_and_activate* is true, the constructor automatically attempts to invoke *server_bind()* and *server_activate()*. The other parameters are passed to the *BaseServer* base class.

class socketserver.**UDPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
This uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The parameters are the same as for *TCPServer*.

class socketserver.**UnixStreamServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
class socketserver.**UnixDatagramServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
These more infrequently used classes are similar to the TCP and UDP classes, but use Unix domain sockets; they're not available on non-Unix platforms. The parameters are the same as for *TCPServer*.

Essas quatro classes processam solicitações *synchronously*; cada solicitação deve ser concluída antes que a próxima solicitação possa ser iniciada. Isso não é adequado se cada solicitação demorar muito para ser concluída, porque exige muita computação ou porque retorna muitos dados que o cliente demora a processar. A solução é criar um processo ou thread separado para lidar com cada solicitação; as classes misturadas *ForkingMixIn* e *ThreadingMixIn* podem ser usadas para oferecer suporte ao comportamento assíncrono.

Creating a server requires several steps. First, you must create a request handler class by subclassing the *BaseRequestHandler* class and overriding its *handle()* method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server's address and the request handler class. It is recommended to use the server in a `with` statement. Then call the *handle_request()* or *serve_forever()* method of the server object to process one or many requests. Finally, call *server_close()* to close the socket (unless you used a `with` statement).

When inheriting from *ThreadingMixIn* for threaded connection behavior, you should explicitly declare how you want your threads to behave on an abrupt shutdown. The *ThreadingMixIn* class defines an attribute *daemon_threads*, which indicates whether or not the server should wait for thread termination. You should set the flag explicitly if you would like threads to behave autonomously; the default is *False*, meaning that Python will not exit until all threads created by *ThreadingMixIn* have exited.

Server classes have the same external methods and attributes, no matter what network protocol they use.

22.21.1 Server Creation Notes

There are five classes in an inheritance diagram, four of which represent synchronous servers of four types:

```
+------------+
| BaseServer |
+------------+
      |
      v
+-----------+         +------------------+
| TCPServer |------->| UnixStreamServer |
```

(continua na próxima página)

(continuação da página anterior)

```
+-----------+         +------------------+
      |
      v
+-----------+         +--------------------+
| UDPServer |-------->| UnixDatagramServer |
+-----------+         +--------------------+
```

Note that *UnixDatagramServer* derives from *UDPServer*, not from *UnixStreamServer* — the only difference between an IP and a Unix stream server is the address family, which is simply repeated in both Unix server classes.

class socketserver.**ForkingMixIn**
class socketserver.**ThreadingMixIn**

Forking and threading versions of each type of server can be created using these mix-in classes. For instance, *ThreadingUDPServer* is created as follows:

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

The mix-in class comes first, since it overrides a method defined in *UDPServer*. Setting the various attributes also changes the behavior of the underlying server mechanism.

ForkingMixIn and the Forking classes mentioned below are only available on POSIX platforms that support *fork()*.

`socketserver.ForkingMixIn.server_close()` waits until all child processes complete, except if `socketserver.ForkingMixIn.block_on_close` attribute is false.

`socketserver.ThreadingMixIn.server_close()` waits until all non-daemon threads complete, except if `socketserver.ThreadingMixIn.block_on_close` attribute is false. Use daemonic threads by setting `ThreadingMixIn.daemon_threads` to `True` to not wait until threads complete.

Alterado na versão 3.7: `socketserver.ForkingMixIn.server_close()` and `socketserver.ThreadingMixIn.server_close()` now waits until all child processes and non-daemonic threads complete. Add a new `socketserver.ForkingMixIn.block_on_close` class attribute to opt-in for the pre-3.7 behaviour.

class socketserver.**ForkingTCPServer**
class socketserver.**ForkingUDPServer**
class socketserver.**ThreadingTCPServer**
class socketserver.**ThreadingUDPServer**

These classes are pre-defined using the mix-in classes.

To implement a service, you must derive a class from *BaseRequestHandler* and redefine its *handle()* method. You can then run various versions of the service by combining one of the server classes with your request handler class. The request handler class must be different for datagram or stream services. This can be hidden by using the handler subclasses *StreamRequestHandler* or *DatagramRequestHandler*.

Of course, you still have to use your head! For instance, it makes no sense to use a forking server if the service contains state in memory that can be modified by different requests, since the modifications in the child process would never reach the initial state kept in the parent process and passed to each child. In this case, you can use a threading server, but you will probably have to use locks to protect the integrity of the shared data.

On the other hand, if you are building an HTTP server where all data is stored externally (for instance, in the file system), a synchronous class will essentially render the service "deaf" while one request is being handled – which may be for a very long time if a client is slow to receive all the data it has requested. Here a threading or forking server is appropriate.

In some cases, it may be appropriate to process part of a request synchronously, but to finish processing in a forked child depending on the request data. This can be implemented by using a synchronous server and doing an explicit fork in the

request handler class *handle()* method.

Another approach to handling multiple simultaneous requests in an environment that supports neither threads nor *fork()* (or where these are too expensive or inappropriate for the service) is to maintain an explicit table of partially finished requests and to use *selectors* to decide which request to work on next (or whether to handle a new incoming request). This is particularly important for stream services where each client can potentially be connected for a long time (if threads or subprocesses cannot be used). See *asyncore* for another way to manage this.

22.21.2 Objetos Server

class socketserver.**BaseServer**(*server_address*, *RequestHandlerClass*)

This is the superclass of all Server objects in the module. It defines the interface, given below, but does not implement most of the methods, which is done in subclasses. The two parameters are stored in the respective *server_address* and *RequestHandlerClass* attributes.

fileno()

Return an integer file descriptor for the socket on which the server is listening. This function is most commonly passed to *selectors*, to allow monitoring multiple servers in the same process.

handle_request()

Process a single request. This function calls the following methods in order: *get_request()*, *verify_request()*, and *process_request()*. If the user-provided *handle()* method of the handler class raises an exception, the server's *handle_error()* method will be called. If no request is received within *timeout* seconds, *handle_timeout()* will be called and *handle_request()* will return.

serve_forever(*poll_interval=0.5*)

Handle requests until an explicit *shutdown()* request. Poll for shutdown every *poll_interval* seconds. Ignores the *timeout* attribute. It also calls *service_actions()*, which may be used by a subclass or mixin to provide actions specific to a given service. For example, the *ForkingMixIn* class uses *service_actions()* to clean up zombie child processes.

Alterado na versão 3.3: Added `service_actions` call to the `serve_forever` method.

service_actions()

This is called in the *serve_forever()* loop. This method can be overridden by subclasses or mixin classes to perform actions specific to a given service, such as cleanup actions.

Novo na versão 3.3.

shutdown()

Tell the *serve_forever()* loop to stop and wait until it does. *shutdown()* must be called while *serve_forever()* is running in a different thread otherwise it will deadlock.

server_close()

Clean up the server. May be overridden.

address_family

The family of protocols to which the server's socket belongs. Common examples are *socket.AF_INET* and *socket.AF_UNIX*.

RequestHandlerClass

The user-provided request handler class; an instance of this class is created for each request.

server_address

The address on which the server is listening. The format of addresses varies depending on the protocol family; see the documentation for the *socket* module for details. For Internet protocols, this is a tuple containing a string giving the address, and an integer port number: `('127.0.0.1', 80)`, for example.

socket
O objeto soquete no qual o servidor ouve para solicitações recebidas.

The server classes support the following class variables:

allow_reuse_address
Whether the server will allow the reuse of an address. This defaults to *False*, and can be set in subclasses to change the policy.

request_queue_size
The size of the request queue. If it takes a long time to process a single request, any requests that arrive while the server is busy are placed into a queue, up to *request_queue_size* requests. Once the queue is full, further requests from clients will get a "Connection denied" error. The default value is usually 5, but this can be overridden by subclasses.

socket_type
The type of socket used by the server; *socket.SOCK_STREAM* and *socket.SOCK_DGRAM* are two common values.

timeout
Timeout duration, measured in seconds, or *None* if no timeout is desired. If *handle_request()* receives no incoming requests within the timeout period, the *handle_timeout()* method is called.

There are various server methods that can be overridden by subclasses of base server classes like *TCPServer*; these methods aren't useful to external users of the server object.

finish_request(*request*, *client_address*)
Actually processes the request by instantiating *RequestHandlerClass* and calling its *handle()* method.

get_request()
Must accept a request from the socket, and return a 2-tuple containing the *new* socket object to be used to communicate with the client, and the client's address.

handle_error(*request*, *client_address*)
This function is called if the *handle()* method of a *RequestHandlerClass* instance raises an exception. The default action is to print the traceback to standard error and continue handling further requests.

Alterado na versão 3.6: Now only called for exceptions derived from the *Exception* class.

handle_timeout()
This function is called when the *timeout* attribute has been set to a value other than *None* and the timeout period has passed with no requests being received. The default action for forking servers is to collect the status of any child processes that have exited, while in threading servers this method does nothing.

process_request(*request*, *client_address*)
Calls *finish_request()* to create an instance of the *RequestHandlerClass*. If desired, this function can create a new process or thread to handle the request; the *ForkingMixIn* and *ThreadingMixIn* classes do this.

server_activate()
Called by the server's constructor to activate the server. The default behavior for a TCP server just invokes *listen()* on the server's socket. May be overridden.

server_bind()
Called by the server's constructor to bind the socket to the desired address. May be overridden.

verify_request(*request*, *client_address*)
Must return a Boolean value; if the value is *True*, the request will be processed, and if it's *False*, the request will be denied. This function can be overridden to implement access controls for a server. The default implementation always returns *True*.

Alterado na versão 3.6: Support for the *context manager* protocol was added. Exiting the context manager is equivalent to calling *server_close()*.

22.21.3 Request Handler Objects

class socketserver.**BaseRequestHandler**

This is the superclass of all request handler objects. It defines the interface, given below. A concrete request handler subclass must define a new *handle()* method, and can override any of the other methods. A new instance of the subclass is created for each request.

setup()

Called before the *handle()* method to perform any initialization actions required. The default implementation does nothing.

handle()

This function must do all the work required to service a request. The default implementation does nothing. Several instance attributes are available to it; the request is available as `self.request`; the client address as `self.client_address`; and the server instance as `self.server`, in case it needs access to per-server information.

The type of `self.request` is different for datagram or stream services. For stream services, `self.request` is a socket object; for datagram services, `self.request` is a pair of string and socket.

finish()

Called after the *handle()* method to perform any clean-up actions required. The default implementation does nothing. If *setup()* raises an exception, this function will not be called.

class socketserver.**StreamRequestHandler**

class socketserver.**DatagramRequestHandler**

These *BaseRequestHandler* subclasses override the *setup()* and *finish()* methods, and provide `self.rfile` and `self.wfile` attributes. The `self.rfile` and `self.wfile` attributes can be read or written, respectively, to get the request data or return data to the client.

The `rfile` attributes of both classes support the *io.BufferedIOBase* readable interface, and `DatagramRequestHandler.wfile` supports the *io.BufferedIOBase* writable interface.

Alterado na versão 3.6: `StreamRequestHandler.wfile` also supports the *io.BufferedIOBase* writable interface.

22.21.4 Exemplos

socketserver.TCPServer Example

This is the server side:

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """
```

(continua na próxima página)

(continuação da página anterior)

```
    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        # Activate the server; this will keep running until you
        # interrupt the program with Ctrl-C
        server.serve_forever()
```

An alternative request handler class that makes use of streams (file-like objects that simplify communication by providing the standard file interface):

```
class MyTCPHandler(socketserver.StreamRequestHandler):

    def handle(self):
        # self.rfile is a file-like object created by the handler;
        # we can now use e.g. readline() instead of raw recv() calls
        self.data = self.rfile.readline().strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # Likewise, self.wfile is a file-like object used to write back
        # to the client
        self.wfile.write(self.data.upper())
```

The difference is that the `readline()` call in the second handler will call `recv()` multiple times until it encounters a newline character, while the single `recv()` call in the first handler will just return what has been sent from the client in one `sendall()` call.

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data + "\n", "utf-8"))

    # Receive data from the server and shut down
    received = str(sock.recv(1024), "utf-8")

print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

The output of the example should look something like this:

Server:

```
$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'
```

Cliente:

```
$ python TCPClient.py hello world with TCP
Sent:     hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent:     python is nice
Received: PYTHON IS NICE
```

Exemplo `socketserver.UDPServer`

This is the server side:

```
import socketserver

class MyUDPHandler(socketserver.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
    when sending data back via sendto().
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print("{} wrote:".format(self.client_address[0]))
        print(data)
        socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as server:
        server.serve_forever()
```

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
# Instead, data is directly sent to the recipient via sendto().
```

(continua na próxima página)

(continuação da página anterior)

```
sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")

print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

The output of the example should look exactly like for the TCP server example.

Asynchronous Mixins

To build asynchronous handlers, use the *ThreadingMixIn* and *ForkingMixIn* classes.

An example for the *ThreadingMixIn* class:

```
import socket
import threading
import socketserver

class ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024), 'ascii')
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data), 'ascii')
        self.request.sendall(response)

class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    pass

def client(ip, port, message):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((ip, port))
        sock.sendall(bytes(message, 'ascii'))
        response = str(sock.recv(1024), 'ascii')
        print("Received: {}".format(response))

if __name__ == "__main__":
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    with server:
        ip, port = server.server_address

        # Start a thread with the server -- that thread will then start one
        # more thread for each request
        server_thread = threading.Thread(target=server.serve_forever)
        # Exit the server thread when the main thread terminates
        server_thread.daemon = True
        server_thread.start()
        print("Server loop running in thread:", server_thread.name)

        client(ip, port, "Hello World 1")
        client(ip, port, "Hello World 2")
        client(ip, port, "Hello World 3")
```

(continua na próxima página)

(continuação da página anterior)

```
        server.shutdown()
```

The output of the example should look something like this:

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

The *ForkingMixIn* class is used in the same way, except that the server will spawn a new process for each request. Available only on POSIX platforms that support *fork()*.

22.22 http.server — servidores HTTP

Código Fonte: Lib/http/server.py

This module defines classes for implementing HTTP servers (Web servers).

Aviso: *http.server* is not recommended for production. It only implements *basic security checks*.

One class, *HTTPServer*, is a *socketserver.TCPServer* subclass. It creates and listens at the HTTP socket, dispatching the requests to a handler. Code to create and run the server looks like this:

```
def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class http.server.**HTTPServer**(*server_address*, *RequestHandlerClass*)

This class builds on the *TCPServer* class by storing the server address as instance variables named `server_name` and `server_port`. The server is accessible by the handler, typically through the handler's `server` instance variable.

class http.server.**ThreadingHTTPServer**(*server_address*, *RequestHandlerClass*)

This class is identical to HTTPServer but uses threads to handle requests by using the *ThreadingMixIn*. This is useful to handle web browsers pre-opening sockets, on which *HTTPServer* would wait indefinitely.

Novo na versão 3.7.

The *HTTPServer* and *ThreadingHTTPServer* must be given a *RequestHandlerClass* on instantiation, of which this module provides three different variants:

class http.server.**BaseHTTPRequestHandler**(*request*, *client_address*, *server*)

This class is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). *BaseHTTPRequestHandler* provides a number of class and instance variables, and methods for use by subclasses.

The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method `SPAM`, the `do_SPAM()` method will be

called with no arguments. All of the relevant information is stored in instance variables of the handler. Subclasses should not need to override or extend the `__init__()` method.

BaseHTTPRequestHandler tem os seguintes atributos de instância:

client_address
Contains a tuple of the form `(host, port)` referring to the client's address.

server
Contains the server instance.

close_connection
Boolean that should be set before *handle_one_request()* returns, indicating if another request may be expected, or if the connection should be shut down.

requestline
Contains the string representation of the HTTP request line. The terminating CRLF is stripped. This attribute should be set by *handle_one_request()*. If no valid request line was processed, it should be set to the empty string.

command
Contains the command (request type). For example, `'GET'`.

path
Contém o caminho solicitado.

request_version
Contains the version string from the request. For example, `'HTTP/1.0'`.

headers
Holds an instance of the class specified by the *MessageClass* class variable. This instance parses and manages the headers in the HTTP request. The `parse_headers()` function from *http.client* is used to parse the headers and it requires that the HTTP request provide a valid **RFC 2822** style header.

rfile
An *io.BufferedIOBase* input stream, ready to read from the start of the optional input data.

wfile
Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream in order to achieve successful interoperation with HTTP clients.

Alterado na versão 3.6: This is an *io.BufferedIOBase* stream.

BaseHTTPRequestHandler tems os seguintes atributos:

server_version
Specifies the server software version. You may want to override this. The format is multiple whitespace-separated strings, where each string is of the form name[/version]. For example, `'BaseHTTP/0.2'`.

sys_version
Contains the Python system version, in a form usable by the *version_string* method and the *server_version* class variable. For example, `'Python/1.4'`.

error_message_format
Specifies a format string that should be used by *send_error()* method for building an error response to the client. The string is filled by default with variables from *responses* based on the status code that passed to *send_error()*.

error_content_type
Specifies the Content-Type HTTP header of error responses sent to the client. The default value is `'text/html'`.

protocol_version
This specifies the HTTP protocol version used in responses. If set to `'HTTP/1.1'`, the server will permit HTTP persistent connections; however, your server *must* then include an accurate `Content-Length` header (using *send_header()*) in all of its responses to clients. For backwards compatibility, the setting defaults to `'HTTP/1.0'`.

MessageClass
Specifies an *email.message.Message*-like class to parse HTTP headers. Typically, this is not overridden, and it defaults to `http.client.HTTPMessage`.

responses
This attribute contains a mapping of error code integers to two-element tuples containing a short and long message. For example, `{code: (shortmessage, longmessage)}`. The *shortmessage* is usually used as the *message* key in an error response, and *longmessage* as the *explain* key. It is used by *send_response_only()* and *send_error()* methods.

Uma instância de *BaseHTTPRequestHandler* têm os seguintes métodos:

handle()
Calls *handle_one_request()* once (or, if persistent connections are enabled, multiple times) to handle incoming HTTP requests. You should never need to override it; instead, implement appropriate `do_*()` methods.

handle_one_request()
This method will parse and dispatch the request to the appropriate `do_*()` method. You should never need to override it.

handle_expect_100()
When a HTTP/1.1 compliant server receives an `Expect: 100-continue` request header it responds back with a `100 Continue` followed by `200 OK` headers. This method can be overridden to raise an error if the server does not want the client to continue. For e.g. server can chose to send `417 Expectation Failed` as a response header and `return False`.

Novo na versão 3.2.

send_error(*code*, *message=None*, *explain=None*)
Sends and logs a complete error reply to the client. The numeric *code* specifies the HTTP error code, with *message* as an optional, short, human readable description of the error. The *explain* argument can be used to provide more detailed information about the error; it will be formatted using the *error_message_format* attribute and emitted, after a complete set of headers, as the response body. The *responses* attribute holds the default values for *message* and *explain* that will be used if no value is provided; for unknown codes the default value for both is the string `???`. The body will be empty if the method is HEAD or the response code is one of the following: `1xx`, `204 No Content`, `205 Reset Content`, `304 Not Modified`.

Alterado na versão 3.4: The error response includes a Content-Length header. Added the *explain* argument.

send_response(*code*, *message=None*)
Adds a response header to the headers buffer and logs the accepted request. The HTTP response line is written to the internal buffer, followed by *Server* and *Date* headers. The values for these two headers are picked up from the *version_string()* and *date_time_string()* methods, respectively. If the server does not intend to send any other headers using the *send_header()* method, then *send_response()* should be followed by an *end_headers()* call.

Alterado na versão 3.3: Headers are stored to an internal buffer and *end_headers()* needs to be called explicitly.

send_header(*keyword*, *value*)
Adds the HTTP header to an internal buffer which will be written to the output stream when either *end_headers()* or *flush_headers()* is invoked. *keyword* should specify the header keyword, with

value specifying its value. Note that, after the send_header calls are done, *end_headers()* MUST BE called in order to complete the operation.

Alterado na versão 3.2: Headers are stored in an internal buffer.

send_response_only(*code*, *message=None*)
Sends the response header only, used for the purposes when `100 Continue` response is sent by the server to the client. The headers not buffered and sent directly the output stream.If the *message* is not specified, the HTTP message corresponding the response *code* is sent.

Novo na versão 3.2.

end_headers()
Adds a blank line (indicating the end of the HTTP headers in the response) to the headers buffer and calls *flush_headers()*.

Alterado na versão 3.2: The buffered headers are written to the output stream.

flush_headers()
Finally send the headers to the output stream and flush the internal headers buffer.

Novo na versão 3.3.

log_request(*code='-'*, *size='-'*)
Logs an accepted (successful) request. *code* should specify the numeric HTTP code associated with the response. If a size of the response is available, then it should be passed as the *size* parameter.

log_error(...)
Logs an error when a request cannot be fulfilled. By default, it passes the message to *log_message()*, so it takes the same arguments (*format* and additional values).

log_message(*format*, ...)
Logs an arbitrary message to `sys.stderr`. This is typically overridden to create custom error logging mechanisms. The *format* argument is a standard printf-style format string, where the additional arguments to *log_message()* are applied as inputs to the formatting. The client ip address and current date and time are prefixed to every message logged.

version_string()
Returns the server software's version string. This is a combination of the *server_version* and *sys_version* attributes.

date_time_string(*timestamp=None*)
Returns the date and time given by *timestamp* (which must be `None` or in the format returned by *time.time()*), formatted for a message header. If *timestamp* is omitted, it uses the current date and time.

The result looks like `'Sun, 06 Nov 1994 08:49:37 GMT'`.

log_date_time_string()
Returns the current date and time, formatted for logging.

address_string()
Retorna o endereço do cliente.

Alterado na versão 3.3: Previously, a name lookup was performed. To avoid name resolution delays, it now always returns the IP address.

class http.server.**SimpleHTTPRequestHandler**(*request*, *client_address*, *server*, *directory=None*)
This class serves files from the current directory and below, directly mapping the directory structure to HTTP requests.

Muito do trabalho, como analisar o pedido, é feito pela classe base *BaseHTTPRequestHandler*. Esta classe implementa as funções *do_GET()* e *do_HEAD()*.

The following are defined as class-level attributes of *SimpleHTTPRequestHandler*:

server_version
This will be "SimpleHTTP/" + __version__, where __version__ is defined at the module level.

extensions_map
A dictionary mapping suffixes into MIME types. The default is signified by an empty string, and is considered to be application/octet-stream. The mapping is used case-insensitively, and so should contain only lower-cased keys.

directory
If not specified, the directory to serve is the current working directory.

The *SimpleHTTPRequestHandler* class defines the following methods:

do_HEAD()
This method serves the 'HEAD' request type: it sends the headers it would send for the equivalent GET request. See the *do_GET()* method for a more complete explanation of the possible headers.

do_GET()
The request is mapped to a local file by interpreting the request as a path relative to the current working directory.

If the request was mapped to a directory, the directory is checked for a file named index.html or index.htm (in that order). If found, the file's contents are returned; otherwise a directory listing is generated by calling the list_directory() method. This method uses *os.listdir()* to scan the directory, and returns a 404 error response if the *listdir()* fails.

If the request was mapped to a file, it is opened. Any *OSError* exception in opening the requested file is mapped to a 404, 'File not found' error. If there was a 'If-Modified-Since' header in the request, and the file was not modified after this time, a 304, 'Not Modified' response is sent. Otherwise, the content type is guessed by calling the guess_type() method, which in turn uses the *extensions_map* variable, and the file contents are returned.

A 'Content-type:' header with the guessed content type is output, followed by a 'Content-Length:' header with the file's size and a 'Last-Modified:' header with the file's modification time.

Then follows a blank line signifying the end of the headers, and then the contents of the file are output. If the file's MIME type starts with text/ the file is opened in text mode; otherwise binary mode is used.

For example usage, see the implementation of the *test()* function invocation in the *http.server* module.

Alterado na versão 3.7: Support of the 'If-Modified-Since' header.

The *SimpleHTTPRequestHandler* class can be used in the following manner in order to create a very basic webserver serving files relative to the current directory:

```
import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
    print("serving at port", PORT)
    httpd.serve_forever()
```

http.server can also be invoked directly using the `-m` switch of the interpreter with a `port number` argument. Similar to the previous example, this serves files relative to the current directory:

```
python -m http.server 8000
```

By default, server binds itself to all interfaces. The option `-b/--bind` specifies a specific address to which it should bind. For example, the following command causes the server to bind to localhost only:

```
python -m http.server 8000 --bind 127.0.0.1
```

Novo na versão 3.4: `--bind` argument was introduced.

By default, server uses the current directory. The option `-d/--directory` specifies a directory to which it should serve the files. For example, the following command uses a specific directory:

```
python -m http.server --directory /tmp/
```

Novo na versão 3.7: `--directory` specify alternate directory

class http.server.**CGIHTTPRequestHandler**(*request*, *client_address*, *server*)

This class is used to serve either files or output of CGI scripts from the current directory and below. Note that mapping HTTP hierarchic structure to local directory structure is exactly as in *SimpleHTTPRequestHandler*.

Nota: CGI scripts run by the *CGIHTTPRequestHandler* class cannot execute redirects (HTTP code 302), because code 200 (script output follows) is sent prior to execution of the CGI script. This pre-empts the status code.

The class will however, run the CGI script, instead of serving it as a file, if it guesses it to be a CGI script. Only directory-based CGI are used — the other common server configuration is to treat special extensions as denoting CGI scripts.

The `do_GET()` and `do_HEAD()` functions are modified to run CGI scripts and serve the output, instead of serving files, if the request leads to somewhere below the `cgi_directories` path.

The *CGIHTTPRequestHandler* defines the following data member:

cgi_directories

This defaults to `['/cgi-bin', '/htbin']` and describes directories to treat as containing CGI scripts.

The *CGIHTTPRequestHandler* defines the following method:

do_POST()

This method serves the `'POST'` request type, only allowed for CGI scripts. Error 501, "Can only POST to CGI scripts", is output when trying to POST to a non-CGI url.

Note that CGI scripts will be run with UID of user nobody, for security reasons. Problems with the CGI script will be translated to error 403.

CGIHTTPRequestHandler can be enabled in the command line by passing the `--cgi` option:

```
python -m http.server --cgi 8000
```

22.22.1 Security Considerations

SimpleHTTPRequestHandler will follow symbolic links when handling requests, this makes it possible for files outside of the specified directory to be served.

Earlier versions of Python did not scrub control characters from the log messages emitted to stderr from `python -m http.server` or the default *BaseHTTPRequestHandler* `.log_message` implementation. This could allow remote clients connecting to your server to send nefarious control codes to your terminal.

Novo na versão 3.7.16: scrubbing control characters from log messages

22.23 `http.cookies` — Gerenciadores de estado HTTP

Código Fonte: Lib/http/cookies.py

The *http.cookies* module defines classes for abstracting the concept of cookies, an HTTP state management mechanism. It supports both simple string-only cookies, and provides an abstraction for having any serializable data-type as cookie value.

The module formerly strictly applied the parsing rules described in the **RFC 2109** and **RFC 2068** specifications. It has since been discovered that MSIE 3.0x doesn't follow the character rules outlined in those specs and also many current day browsers and servers have relaxed parsing rules when comes to Cookie handling. As a result, the parsing rules used are a bit less strict.

The character set, *string.ascii_letters*, *string.digits* and `!#$%&'*+-.^_`|~:` denote the set of valid characters allowed by this module in Cookie name (as *key*).

Alterado na versão 3.3: Allowed ':' as a valid Cookie name character.

Nota: On encountering an invalid cookie, *CookieError* is raised, so if your cookie data comes from a browser you should always prepare for invalid data and catch *CookieError* on parsing.

---

exception `http.cookies.`**CookieError**
Exception failing because of **RFC 2109** invalidity: incorrect attributes, incorrect *Set-Cookie* header, etc.

class `http.cookies.`**BaseCookie**(*[input]*)
This class is a dictionary-like object whose keys are strings and whose values are *Morsel* instances. Note that upon setting a key to a value, the value is first converted to a *Morsel* containing the key and the value.

If *input* is given, it is passed to the *load()* method.

class `http.cookies.`**SimpleCookie**(*[input]*)
This class derives from *BaseCookie* and overrides `value_decode()` and `value_encode()`. SimpleCookie supports strings as cookie values. When setting the value, SimpleCookie calls the builtin *str()* to convert the value to a string. Values received from HTTP are kept as strings.

Ver também:

Módulo *http.cookiejar* HTTP cookie handling for web *clients*. The *http.cookiejar* and *http.cookies* modules do not depend on each other.

RFC 2109 - HTTP State Management Mechanism This is the state management specification implemented by this module.

22.23.1 Objetos Cookie

BaseCookie.**value_decode**(*val*)
Return a tuple `(real_value, coded_value)` from a string representation. `real_value` can be any type. This method does no decoding in *BaseCookie* — it exists so it can be overridden.

BaseCookie.**value_encode**(*val*)
Return a tuple `(real_value, coded_value)`. *val* can be any type, but `coded_value` will always be converted to a string. This method does no encoding in *BaseCookie* — it exists so it can be overridden.

In general, it should be the case that *value_encode()* and *value_decode()* are inverses on the range of *value_decode*.

BaseCookie.**output**(*attrs=None*, *header='Set-Cookie:'*, *sep='\r\n'*)
Return a string representation suitable to be sent as HTTP headers. *attrs* and *header* are sent to each *Morsel*'s *output()* method. *sep* is used to join the headers together, and is by default the combination `'\r\n'` (CRLF).

BaseCookie.**js_output**(*attrs=None*)
Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP headers was sent.

The meaning for *attrs* is the same as in *output()*.

BaseCookie.**load**(*rawdata*)
If *rawdata* is a string, parse it as an `HTTP_COOKIE` and add the values found there as *Morsel*s. If it is a dictionary, it is equivalent to:

```
for k, v in rawdata.items():
    cookie[k] = v
```

22.23.2 Objetos Morsel

class http.cookies.**Morsel**
Abstract a key/value pair, which has some **RFC 2109** attributes.

Morsels are dictionary-like objects, whose set of keys is constant — the valid **RFC 2109** attributes, which are

- `expires`
- `path`
- `comment`
- `domain`
- `max-age`
- `secure`
- `version`
- `httponly`

The attribute `httponly` specifies that the cookie is only transferred in HTTP requests, and is not accessible through JavaScript. This is intended to mitigate some forms of cross-site scripting.

The keys are case-insensitive and their default value is `''`.

Alterado na versão 3.5: `__eq__()` now takes *key* and *value* into account.

Alterado na versão 3.7: Attributes *key*, *value* and *coded_value* are read-only. Use *set()* for setting them.

Morsel.**value**
O valor do cookie.

Morsel.**coded_value**
The encoded value of the cookie — this is what should be sent.

Morsel.**key**
O nome do cookie.

Morsel.**set**(*key*, *value*, *coded_value*)
Set the *key*, *value* and *coded_value* attributes.

Morsel.**isReservedKey**(*K*)
Whether *K* is a member of the set of keys of a *Morsel*.

Morsel.**output**(*attrs=None*, *header='Set-Cookie:'*)
Return a string representation of the Morsel, suitable to be sent as an HTTP header. By default, all the attributes are included, unless *attrs* is given, in which case it should be a list of attributes to use. *header* is by default `"Set-Cookie:"`.

Morsel.**js_output**(*attrs=None*)
Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP header was sent.

The meaning for *attrs* is the same as in *output()*.

Morsel.**OutputString**(*attrs=None*)
Return a string representing the Morsel, without any surrounding HTTP or JavaScript.

The meaning for *attrs* is the same as in *output()*.

Morsel.**update**(*values*)
Update the values in the Morsel dictionary with the values in the dictionary *values*. Raise an error if any of the keys in the *values* dict is not a valid **RFC 2109** attribute.

Alterado na versão 3.5: an error is raised for invalid keys.

Morsel.**copy**(*value*)
Return a shallow copy of the Morsel object.

Alterado na versão 3.5: return a Morsel object instead of a dict.

Morsel.**setdefault**(*key*, *value=None*)
Raise an error if key is not a valid **RFC 2109** attribute, otherwise behave the same as *dict.setdefault()*.

22.23.3 Exemplo

The following example demonstrates how to use the *http.cookies* module.

```
>>> from http import cookies
>>> C = cookies.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print(C) # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print(C.output()) # same thing
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C = cookies.SimpleCookie()
```

(continua na próxima página)

(continuação da página anterior)

```
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print(C.output(header="Cookie:"))
Cookie: rocky=road; Path=/cookie
>>> print(C.output(attrs=[], header="Cookie:"))
Cookie: rocky=road
>>> C = cookies.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # load from a string (HTTP header)
>>> print(C)
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = cookies.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\"Loves\\"; fudge=\\012;";')
>>> print(C)
Set-Cookie: keebler="E=everybody; L=\"Loves\"; fudge=\012;"
>>> C = cookies.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print(C)
Set-Cookie: oreo=doublestuff; Path=/
>>> C = cookies.SimpleCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = cookies.SimpleCookie()
>>> C["number"] = 7 # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print(C)
Set-Cookie: number=7
Set-Cookie: string=seven
```

22.24 `http.cookiejar` — Cookie handling for HTTP clients

Código Fonte: Lib/http/cookiejar.py

The *`http.cookiejar`* module defines classes for automatic handling of HTTP cookies. It is useful for accessing web sites that require small pieces of data – *cookies* – to be set on the client machine by an HTTP response from a web server, and then returned to the server in later HTTP requests.

Both the regular Netscape cookie protocol and the protocol defined by **RFC 2965** are handled. RFC 2965 handling is switched off by default. **RFC 2109** cookies are parsed as Netscape cookies and subsequently treated either as Netscape or RFC 2965 cookies according to the 'policy' in effect. Note that the great majority of cookies on the Internet are Netscape cookies. *`http.cookiejar`* attempts to follow the de-facto Netscape cookie protocol (which differs substantially from that set out in the original Netscape specification), including taking note of the `max-age` and `port` cookie-attributes introduced with RFC 2965.

Nota: The various named parameters found in *`Set-Cookie`* and *`Set-Cookie2`* headers (eg. `domain` and `expires`) are conventionally referred to as *attributes*. To distinguish them from Python attributes, the documentation

for this module uses the term *cookie-attribute* instead.

The module defines the following exception:

exception http.cookiejar.**LoadError**
Instances of `FileCookieJar` raise this exception on failure to load cookies from a file. `LoadError` is a subclass of `OSError`.

Alterado na versão 3.3: LoadError was made a subclass of `OSError` instead of `IOError`.

The following classes are provided:

class http.cookiejar.**CookieJar**(*policy=None*)
policy is an object implementing the `CookiePolicy` interface.

The `CookieJar` class stores HTTP cookies. It extracts cookies from HTTP requests, and returns them in HTTP responses. `CookieJar` instances automatically expire contained cookies when necessary. Subclasses are also responsible for storing and retrieving cookies from a file or database.

class http.cookiejar.**FileCookieJar**(*filename*, *delayload=None*, *policy=None*)
policy is an object implementing the `CookiePolicy` interface. For the other arguments, see the documentation for the corresponding attributes.

A `CookieJar` which can load cookies from, and perhaps save cookies to, a file on disk. Cookies are **NOT** loaded from the named file until either the `load()` or `revert()` method is called. Subclasses of this class are documented in section *FileCookieJar subclasses and co-operation with web browsers*.

class http.cookiejar.**CookiePolicy**
This class is responsible for deciding whether each cookie should be accepted from / returned to the server.

class http.cookiejar.**DefaultCookiePolicy**(*blocked_domains=None*, *allowed_domains=None*, *netscape=True*, *rfc2965=False*, *rfc2109_as_netscape=None*, *hide_cookie2=False*, *strict_domain=False*, *strict_rfc2965_unverifiable=True*, *strict_ns_unverifiable=False*, *strict_ns_domain=DefaultCookiePolicy.DomainLiberal*, *strict_ns_set_initial_dollar=False*, *strict_ns_set_path=False*)
Constructor arguments should be passed as keyword arguments only. *blocked_domains* is a sequence of domain names that we never accept cookies from, nor return cookies to. *allowed_domains* if not `None`, this is a sequence of the only domains for which we accept and return cookies. For all other arguments, see the documentation for `CookiePolicy` and `DefaultCookiePolicy` objects.

`DefaultCookiePolicy` implements the standard accept / reject rules for Netscape and **RFC 2965** cookies. By default, **RFC 2109** cookies (ie. cookies received in a `Set-Cookie` header with a version cookie-attribute of 1) are treated according to the RFC 2965 rules. However, if RFC 2965 handling is turned off or `rfc2109_as_netscape` is `True`, RFC 2109 cookies are 'downgraded' by the `CookieJar` instance to Netscape cookies, by setting the `version` attribute of the `Cookie` instance to 0. `DefaultCookiePolicy` also provides some parameters to allow some fine-tuning of policy.

class http.cookiejar.**Cookie**
This class represents Netscape, **RFC 2109** and **RFC 2965** cookies. It is not expected that users of `http.cookiejar` construct their own `Cookie` instances. Instead, if necessary, call `make_cookies()` on a `CookieJar` instance.

Ver também:

Módulo `urllib.request` URL opening with automatic cookie handling.

Module *http.cookies* HTTP cookie classes, principally useful for server-side code. The *http.cookiejar* and *http.cookies* modules do not depend on each other.

https://curl.haxx.se/rfc/cookie_spec.html The specification of the original Netscape cookie protocol. Though this is still the dominant protocol, the 'Netscape cookie protocol' implemented by all the major browsers (and *http.cookiejar*) only bears a passing resemblance to the one sketched out in `cookie_spec.html`.

RFC 2109 - HTTP State Management Mechanism Obsoleted by **RFC 2965**. Uses *Set-Cookie* with version=1.

RFC 2965 - HTTP State Management Mechanism The Netscape protocol with the bugs fixed. Uses *Set-Cookie2* in place of *Set-Cookie*. Not widely used.

http://kristol.org/cookie/errata.html Unfinished errata to **RFC 2965**.

RFC 2964 - Use of HTTP State Management

22.24.1 CookieJar and FileCookieJar Objects

CookieJar objects support the *iterator* protocol for iterating over contained *Cookie* objects.

CookieJar has the following methods:

`CookieJar.`**`add_cookie_header`**(*request*)
Add correct *Cookie* header to *request*.

If policy allows (ie. the `rfc2965` and `hide_cookie2` attributes of the *CookieJar*'s *CookiePolicy* instance are true and false respectively), the *Cookie2* header is also added when appropriate.

The *request* object (usually a *urllib.request.Request* instance) must support the methods `get_full_url()`, `get_host()`, `get_type()`, `unverifiable()`, `has_header()`, `get_header()`, `header_items()`, `add_unredirected_header()` and `origin_req_host` attribute as documented by *urllib.request*.

Alterado na versão 3.3: *request* object needs `origin_req_host` attribute. Dependency on a deprecated method `get_origin_req_host()` has been removed.

`CookieJar.`**`extract_cookies`**(*response*, *request*)
Extract cookies from HTTP *response* and store them in the *CookieJar*, where allowed by policy.

The *CookieJar* will look for allowable *Set-Cookie* and *Set-Cookie2* headers in the *response* argument, and store cookies as appropriate (subject to the *CookiePolicy.set_ok()* method's approval).

The *response* object (usually the result of a call to *urllib.request.urlopen()*, or similar) should support an `info()` method, which returns an *email.message.Message* instance.

The *request* object (usually a *urllib.request.Request* instance) must support the methods `get_full_url()`, `get_host()`, `unverifiable()`, and `origin_req_host` attribute, as documented by *urllib.request*. The request is used to set default values for cookie-attributes as well as for checking that the cookie is allowed to be set.

Alterado na versão 3.3: *request* object needs `origin_req_host` attribute. Dependency on a deprecated method `get_origin_req_host()` has been removed.

`CookieJar.`**`set_policy`**(*policy*)
Set the *CookiePolicy* instance to be used.

`CookieJar.`**`make_cookies`**(*response*, *request*)
Return sequence of *Cookie* objects extracted from *response* object.

See the documentation for *extract_cookies()* for the interfaces required of the *response* and *request* arguments.

CookieJar.**set_cookie_if_ok**(*cookie*, *request*)
Set a *Cookie* if policy says it's OK to do so.

CookieJar.**set_cookie**(*cookie*)
Set a *Cookie*, without checking with policy to see whether or not it should be set.

CookieJar.**clear**([*domain*[, *path*[, *name*]]])
Clear some cookies.

If invoked without arguments, clear all cookies. If given a single argument, only cookies belonging to that *domain* will be removed. If given two arguments, cookies belonging to the specified *domain* and URL *path* are removed. If given three arguments, then the cookie with the specified *domain*, *path* and *name* is removed.

Raises *KeyError* if no matching cookie exists.

CookieJar.**clear_session_cookies**()
Descarte os cookies de sessão.

Discards all contained cookies that have a true `discard` attribute (usually because they had either no `max-age` or `expires` cookie-attribute, or an explicit `discard` cookie-attribute). For interactive browsers, the end of a session usually corresponds to closing the browser window.

Note that the `save()` method won't save session cookies anyway, unless you ask otherwise by passing a true *ignore_discard* argument.

FileCookieJar implements the following additional methods:

FileCookieJar.**save**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Save cookies to a file.

This base class raises *NotImplementedError*. Subclasses may leave this method unimplemented.

filename is the name of file in which to save cookies. If *filename* is not specified, `self.filename` is used (whose default is the value passed to the constructor, if any); if `self.filename` is *None*, *ValueError* is raised.

ignore_discard: save even cookies set to be discarded. *ignore_expires*: save even cookies that have expired

The file is overwritten if it already exists, thus wiping all the cookies it contains. Saved cookies can be restored later using the *load()* or *revert()* methods.

FileCookieJar.**load**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Load cookies from a file.

Old cookies are kept unless overwritten by newly loaded ones.

Arguments are as for *save()*.

The named file must be in the format understood by the class, or *LoadError* will be raised. Also, *OSError* may be raised, for example if the file does not exist.

Alterado na versão 3.3: *IOError* costumava ser levantado, agora ele é um codinome para *OSError*.

FileCookieJar.**revert**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Clear all cookies and reload cookies from a saved file.

revert() can raise the same exceptions as *load()*. If there is a failure, the object's state will not be altered.

FileCookieJar instances have the following public attributes:

FileCookieJar.**filename**
Filename of default file in which to keep cookies. This attribute may be assigned to.

FileCookieJar.**delayload**
If true, load cookies lazily from disk. This attribute should not be assigned to. This is only a hint, since this only

affects performance, not behaviour (unless the cookies on disk are changing). A *CookieJar* object may ignore it. None of the *FileCookieJar* classes included in the standard library lazily loads cookies.

22.24.2 FileCookieJar subclasses and co-operation with web browsers

The following *CookieJar* subclasses are provided for reading and writing.

class http.cookiejar.**MozillaCookieJar**(*filename*, *delayload=None*, *policy=None*)
A *FileCookieJar* that can load from and save cookies to disk in the Mozilla `cookies.txt` file format (which is also used by the Lynx and Netscape browsers).

Nota: This loses information about **RFC 2965** cookies, and also about newer or non-standard cookie-attributes such as `port`.

Aviso: Back up your cookies before saving if you have cookies whose loss / corruption would be inconvenient (there are some subtleties which may lead to slight changes in the file over a load / save round-trip).

Also note that cookies saved while Mozilla is running will get clobbered by Mozilla.

class http.cookiejar.**LWPCookieJar**(*filename*, *delayload=None*, *policy=None*)
A *FileCookieJar* that can load from and save cookies to disk in format compatible with the libwww-perl library's `Set-Cookie3` file format. This is convenient if you want to store cookies in a human-readable file.

22.24.3 Objeto CookiePolicy

Objects implementing the *CookiePolicy* interface have the following methods:

CookiePolicy.**set_ok**(*cookie*, *request*)
Return boolean value indicating whether cookie should be accepted from server.

cookie is a *Cookie* instance. *request* is an object implementing the interface defined by the documentation for *CookieJar.extract_cookies()*.

CookiePolicy.**return_ok**(*cookie*, *request*)
Return boolean value indicating whether cookie should be returned to server.

cookie is a *Cookie* instance. *request* is an object implementing the interface defined by the documentation for *CookieJar.add_cookie_header()*.

CookiePolicy.**domain_return_ok**(*domain*, *request*)
Return `False` if cookies should not be returned, given cookie domain.

This method is an optimization. It removes the need for checking every cookie with a particular domain (which might involve reading many files). Returning true from *domain_return_ok()* and *path_return_ok()* leaves all the work to *return_ok()*.

If *domain_return_ok()* returns true for the cookie domain, *path_return_ok()* is called for the cookie path. Otherwise, *path_return_ok()* and *return_ok()* are never called for that cookie domain. If *path_return_ok()* returns true, *return_ok()* is called with the *Cookie* object itself for a full check. Otherwise, *return_ok()* is never called for that cookie path.

Note that *domain_return_ok()* is called for every *cookie* domain, not just for the *request* domain. For example, the function might be called with both `".example.com"` and `"www.example.com"` if the request domain is `"www.example.com"`. The same goes for *path_return_ok()*.

The *request* argument is as documented for *return_ok()*.

CookiePolicy.**path_return_ok**(*path*, *request*)
Return False if cookies should not be returned, given cookie path.

See the documentation for *domain_return_ok()*.

In addition to implementing the methods above, implementations of the *CookiePolicy* interface must also supply the following attributes, indicating which protocols should be used, and how. All of these attributes may be assigned to.

CookiePolicy.**netscape**
Implement Netscape protocol.

CookiePolicy.**rfc2965**
Implement **RFC 2965** protocol.

CookiePolicy.**hide_cookie2**
Don't add *Cookie2* header to requests (the presence of this header indicates to the server that we understand **RFC 2965** cookies).

The most useful way to define a *CookiePolicy* class is by subclassing from *DefaultCookiePolicy* and overriding some or all of the methods above. *CookiePolicy* itself may be used as a 'null policy' to allow setting and receiving any and all cookies (this is unlikely to be useful).

22.24.4 DefaultCookiePolicy Objects

Implements the standard rules for accepting and returning cookies.

Both **RFC 2965** and Netscape cookies are covered. RFC 2965 handling is switched off by default.

The easiest way to provide your own policy is to override this class and call its methods in your overridden implementations before adding your own additional checks:

```
import http.cookiejar
class MyCookiePolicy(http.cookiejar.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not http.cookiejar.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

In addition to the features required to implement the *CookiePolicy* interface, this class allows you to block and allow domains from setting and receiving cookies. There are also some strictness switches that allow you to tighten up the rather loose Netscape protocol rules a little bit (at the cost of blocking some benign cookies).

A domain blacklist and whitelist is provided (both off by default). Only domains not in the blacklist and present in the whitelist (if the whitelist is active) participate in cookie setting and returning. Use the *blocked_domains* constructor argument, and blocked_domains() and set_blocked_domains() methods (and the corresponding argument and methods for *allowed_domains*). If you set a whitelist, you can turn it off again by setting it to *None*.

Domains in block or allow lists that do not start with a dot must equal the cookie domain to be matched. For example, "example.com" matches a blacklist entry of "example.com", but "www.example.com" does not. Domains that do start with a dot are matched by more specific domains too. For example, both "www.example.com" and "www.coyote.example.com" match ".example.com" (but "example.com" itself does not). IP addresses are an exception, and must match exactly. For example, if blocked_domains contains "192.168.1.2" and ".168.1.2", 192.168.1.2 is blocked, but 193.168.1.2 is not.

DefaultCookiePolicy implements the following additional methods:

DefaultCookiePolicy.**blocked_domains**()
Return the sequence of blocked domains (as a tuple).

DefaultCookiePolicy.**set_blocked_domains**(*blocked_domains*)
Set the sequence of blocked domains.

DefaultCookiePolicy.**is_blocked**(*domain*)
Return whether *domain* is on the blacklist for setting or receiving cookies.

DefaultCookiePolicy.**allowed_domains**()
Return *None*, or the sequence of allowed domains (as a tuple).

DefaultCookiePolicy.**set_allowed_domains**(*allowed_domains*)
Set the sequence of allowed domains, or *None*.

DefaultCookiePolicy.**is_not_allowed**(*domain*)
Return whether *domain* is not on the whitelist for setting or receiving cookies.

DefaultCookiePolicy instances have the following attributes, which are all initialised from the constructor arguments of the same name, and which may all be assigned to.

DefaultCookiePolicy.**rfc2109_as_netscape**
If true, request that the *CookieJar* instance downgrade **RFC 2109** cookies (ie. cookies received in a *Set-Cookie* header with a version cookie-attribute of 1) to Netscape cookies by setting the version attribute of the *Cookie* instance to 0. The default value is *None*, in which case RFC 2109 cookies are downgraded if and only if **RFC 2965** handling is turned off. Therefore, RFC 2109 cookies are downgraded by default.

General strictness switches:

DefaultCookiePolicy.**strict_domain**
Don't allow sites to set two-component domains with country-code top-level domains like `.co.uk`, `.gov.uk`, `.co.nz`.etc. This is far from perfect and isn't guaranteed to work!

RFC 2965 protocol strictness switches:

DefaultCookiePolicy.**strict_rfc2965_unverifiable**
Follow **RFC 2965** rules on unverifiable transactions (usually, an unverifiable transaction is one resulting from a redirect or a request for an image hosted on another site). If this is false, cookies are *never* blocked on the basis of verifiability

Netscape protocol strictness switches:

DefaultCookiePolicy.**strict_ns_unverifiable**
Apply **RFC 2965** rules on unverifiable transactions even to Netscape cookies.

DefaultCookiePolicy.**strict_ns_domain**
Flags indicating how strict to be with domain-matching rules for Netscape cookies. See below for acceptable values.

DefaultCookiePolicy.**strict_ns_set_initial_dollar**
Ignore cookies in Set-Cookie: headers that have names starting with `'$'`.

DefaultCookiePolicy.**strict_ns_set_path**
Don't allow setting cookies whose path doesn't path-match request URI.

`strict_ns_domain` is a collection of flags. Its value is constructed by or-ing together (for example, `DomainStrictNoDots|DomainStrictNonDomain` means both flags are set).

DefaultCookiePolicy.**DomainStrictNoDots**
When setting cookies, the 'host prefix' must not contain a dot (eg. `www.foo.bar.com` can't set a cookie for `.bar.com`, because `www.foo` contains a dot).

DefaultCookiePolicy.**DomainStrictNonDomain**
Cookies that did not explicitly specify a `domain` cookie-attribute can only be returned to a domain equal to the

domain that set the cookie (eg. `spam.example.com` won't be returned cookies from `example.com` that had no `domain` cookie-attribute).

`DefaultCookiePolicy.`**`DomainRFC2965Match`**
When setting cookies, require a full **RFC 2965** domain-match.

The following attributes are provided for convenience, and are the most useful combinations of the above flags:

`DefaultCookiePolicy.`**`DomainLiberal`**
Equivalent to 0 (ie. all of the above Netscape domain strictness flags switched off).

`DefaultCookiePolicy.`**`DomainStrict`**
Equivalente a `DomainStrictNoDots|DomainStrictNonDomain`.

22.24.5 Objetos Cookie

`Cookie` instances have Python attributes roughly corresponding to the standard cookie-attributes specified in the various cookie standards. The correspondence is not one-to-one, because there are complicated rules for assigning default values, because the `max-age` and `expires` cookie-attributes contain equivalent information, and because **RFC 2109** cookies may be 'downgraded' by *`http.cookiejar`* from version 1 to version 0 (Netscape) cookies.

Assignment to these attributes should not be necessary other than in rare circumstances in a *`CookiePolicy`* method. The class does not enforce internal consistency, so you should know what you're doing if you do that.

`Cookie.`**`version`**
Integer or *`None`*. Netscape cookies have *`version`* 0. **RFC 2965** and **RFC 2109** cookies have a `version` cookie-attribute of 1. However, note that *`http.cookiejar`* may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case *`version`* is 0.

`Cookie.`**`name`**
Cookie name (a string).

`Cookie.`**`value`**
Cookie value (a string), or *`None`*.

`Cookie.`**`port`**
String representing a port or a set of ports (eg. '80', or '80,8080'), or *`None`*.

`Cookie.`**`path`**
Cookie path (a string, eg. `'/acme/rocket_launchers'`).

`Cookie.`**`secure`**
`True` if cookie should only be returned over a secure connection.

`Cookie.`**`expires`**
Integer expiry date in seconds since epoch, or *`None`*. See also the *`is_expired()`* method.

`Cookie.`**`discard`**
`True` if this is a session cookie.

`Cookie.`**`comment`**
String comment from the server explaining the function of this cookie, or *`None`*.

`Cookie.`**`comment_url`**
URL linking to a comment from the server explaining the function of this cookie, or *`None`*.

`Cookie.`**`rfc2109`**
`True` if this cookie was received as an **RFC 2109** cookie (ie. the cookie arrived in a *`Set-Cookie`* header, and the value of the Version cookie-attribute in that header was 1). This attribute is provided because *`http.cookiejar`* may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case *`version`* is 0.

`Cookie.`**`port_specified`**
`True` if a port or set of ports was explicitly specified by the server (in the *`Set-Cookie`* / *`Set-Cookie2`* header).

`Cookie.`**`domain_specified`**
`True` if a domain was explicitly specified by the server.

`Cookie.`**`domain_initial_dot`**
`True` if the domain explicitly specified by the server began with a dot (`'.'`).

Cookies may have additional non-standard cookie-attributes. These may be accessed using the following methods:

`Cookie.`**`has_nonstandard_attr`**(*name*)
Return `True` if cookie has the named cookie-attribute.

`Cookie.`**`get_nonstandard_attr`**(*name*, *default=None*)
If cookie has the named cookie-attribute, return its value. Otherwise, return *default*.

`Cookie.`**`set_nonstandard_attr`**(*name*, *value*)
Set the value of the named cookie-attribute.

The *`Cookie`* class also defines the following method:

`Cookie.`**`is_expired`**(*now=None*)
`True` if cookie has passed the time at which the server requested it should expire. If *now* is given (in seconds since the epoch), return whether the cookie has expired at the specified time.

22.24.6 Exemplos

The first example shows the most common usage of *`http.cookiejar`*:

```
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

This example illustrates how to open a URL using your Netscape, Mozilla, or Lynx cookies (assumes Unix/Netscape convention for location of the cookies file):

```
import os, http.cookiejar, urllib.request
cj = http.cookiejar.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

The next example illustrates the use of *`DefaultCookiePolicy`*. Turn on **RFC 2965** cookies, be more strict about domains when setting and returning Netscape cookies, and block some domains from setting cookies or having them returned:

```
import urllib.request
from http.cookiejar import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

22.25 xmlrpc — Módulos de servidor e cliente XMLRPC

XML-RPC é um método de chamada de procedimento remoto que usa XML passado via HTTP como um transporte. Com ele, um cliente pode chamar métodos com parâmetros em um servidor remoto (o servidor é nomeado por um URI) e receber dados estruturados.

`xmlrpc` é um pacote que coleta módulos de servidor e de cliente que implementam o XML-RPC. Os módulos são:

- *xmlrpc.client*
- *xmlrpc.server*

22.26 xmlrpc.client — XML-RPC client access

Código Fonte: Lib/xmlrpc/client.py

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP(S) as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

> **Aviso:** The *xmlrpc.client* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML vulnerabilities*.

Alterado na versão 3.5: For HTTPS URIs, *xmlrpc.client* now performs all the necessary certificate and hostname checks by default.

class xmlrpc.client.**ServerProxy**(*uri*, *transport=None*, *encoding=None*, *verbose=False*, *allow_none=False*, *use_datetime=False*, *use_builtin_types=False*, *, *context=None*)

Alterado na versão 3.3: O sinalizador *use_builtin_types* foi adicionado.

A *ServerProxy* instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance; by default it is an internal `SafeTransport` instance for https: URLs and an internal HTTP `Transport` instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag.

The following parameters govern the use of the returned proxy instance. If *allow_none* is true, the Python constant `None` will be translated into XML; the default behaviour is for `None` to raise a *TypeError*. This is a commonly-used extension to the XML-RPC specification, but isn't supported by all clients and servers; see http://ontosys.com/xml-rpc/extensions.php for a description. The *use_builtin_types* flag can be used to cause date/time values to be presented as *datetime.datetime* objects and binary data to be presented as *bytes* objects; this flag is false by default. *datetime.datetime*, *bytes* and *bytearray* objects may be passed to calls. The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication: `http://user:pass@host:port/path`. The `user:pass` portion will be base64-encoded as an HTTP 'Authorization' header, and sent to the remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password. If an HTTPS URL is provided, *context* may be *ssl.SSLContext* and configures the SSL settings of the underlying HTTPS connection.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.

Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type):

Tipo XML-RPC	Python type
`boolean`	*bool*
`int`, `i1`, `i2`, `i4`, `i8` or `biginteger`	*int* in range from -2147483648 to 2147483647. Values get the `<int>` tag.
`double` or `float`	*float*. Values get the `<double>` tag.
`string`	*str*
`array`	*list* or *tuple* containing conformable elements. Arrays are returned as *lists*.
`struct`	*dict*. Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in; only their *__dict__* attribute is transmitted.
`dateTime.iso8601`	*DateTime* ou *datetime.datetime*. O tipo retornado depende de volumes de sinalizadores *use_builtin_types* e *use_datetime*.
`base64`	*Binary*, *bytes* or *bytearray*. Returned type depends on the value of the *use_builtin_types* flag.
`nil`	The `None` constant. Passing is allowed only if *allow_none* is true.
`bigdecimal`	*decimal.Decimal*. Returned type only.

This is the full set of data types supported by XML-RPC. Method calls may also raise a special *Fault* instance, used to signal XML-RPC server errors, or *ProtocolError* used to signal an error in the HTTP/HTTPS transport layer. Both *Fault* and *ProtocolError* derive from a base class called `Error`. Note that the xmlrpc client module currently does not marshal instances of subclasses of built-in types.

When passing strings, characters special to XML such as `<`, `>`, and `&` will be automatically escaped. However, it's the caller's responsibility to ensure that the string is free of characters that aren't allowed in XML, such as the control characters with ASCII values between 0 and 31 (except, of course, tab, newline and carriage return); failing to do this will result in an XML-RPC request that isn't well-formed XML. If you have to pass arbitrary bytes via XML-RPC, use *bytes* or *bytearray* classes or the *Binary* wrapper class described below.

`Server` is retained as an alias for *ServerProxy* for backwards compatibility. New code should use *ServerProxy*.

Alterado na versão 3.5: Argumento *context* adicionado.

Alterado na versão 3.6: Added support of type tags with prefixes (e.g. `ex:nil`). Added support of unmarshalling additional types used by Apache XML-RPC implementation for numerics: `i1`, `i2`, `i8`, `biginteger`, `float` and `bigdecimal`. See http://ws.apache.org/xmlrpc/types.html for a description.

Ver também:

XML-RPC HOWTO A good description of XML-RPC operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.

XML-RPC Introspection Describes the XML-RPC protocol extension for introspection.

XML-RPC Specification The official specification.

Unofficial XML-RPC Errata Fredrik Lundh's "unofficial errata, intended to clarify certain details in the XML-RPC specification, as well as hint at 'best practices' to use when designing your own XML-RPC implementations."

22.26.1 Objetos ServerProxy

A *ServerProxy* instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a *Fault* or *ProtocolError* object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved `system` attribute:

`ServerProxy.system.`**`listMethods`**`()`
: This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

`ServerProxy.system.`**`methodSignature`**`(`*name*`)`
: This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

  Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

  Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply "string, array". If it expects three integers and returns a string, its signature is "string, int, int, int".

  If no signature is defined for the method, a non-array value is returned. In Python this means that the type of the returned value will be something other than list.

`ServerProxy.system.`**`methodHelp`**`(`*name*`)`
: This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.

Alterado na versão 3.5: Instances of *ServerProxy* support the *context manager* protocol for closing the underlying transport.

A working example follows. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))
```

22.26.2 Objetos DateTime

class xmlrpc.client.**DateTime**

This class may be initialized with seconds since the epoch, a time tuple, an ISO 8601 time/date string, or a *datetime.datetime* instance. It has the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*string*)

Accept a string as the instance's new time value.

encode(*out*)

Write the XML-RPC encoding of this *DateTime* item to the *out* stream object.

It also supports certain of Python's built-in operators through rich comparison and __repr__() methods.

A working example follows. The server code:

```
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(today, "today")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client
import datetime

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```

22.26.3 Objetos Binários

class xmlrpc.client.**Binary**

This class may be initialized from bytes data (which may include NULs). The primary access to the content of a *Binary* object is provided by an attribute:

data

The binary data encapsulated by the *Binary* instance. The data is provided as a *bytes* object.

Binary objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*bytes*)

Accept a base64 *bytes* object and decode it as the instance's new data.

encode(*out*)

Write the XML-RPC base 64 encoding of this binary item to the *out* stream object.

The encoded data will have newlines every 76 characters as per **RFC 2045 section 6.8**, which was the de facto standard base64 specification when the XML-RPC spec was written.

It also supports certain of Python's built-in operators through `__eq__()` and `__ne__()` methods.

Example usage of the binary objects. We're going to transfer an image over XMLRPC:

```
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

The client gets the image and saves it to a file:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

22.26.4 Objetos Fault

class xmlrpc.client.**Fault**

A *Fault* object encapsulates the content of an XML-RPC fault tag. Fault objects have the following attributes:

faultCode

A string indicating the fault type.

faultString

A string containing a diagnostic message associated with the fault.

In the following example we're going to intentionally cause a *Fault* by returning a complex type object. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning a
# complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(add, 'add')

server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
```

22.26.5 Objeto ProtocolError

class xmlrpc.client.**ProtocolError**

A *ProtocolError* object describes a protocol error in the underlying transport layer (such as a 404 'not found' error if the server named by the URI does not exist). It has the following attributes:

url

The URI or URL that triggered the error.

errcode

O código do erro.

errmsg

The error message or diagnostic string.

headers

A dict containing the headers of the HTTP/HTTPS request that triggered the error.

In the following example we're going to intentionally cause a *ProtocolError* by providing an invalid URI:

```
import xmlrpc.client

# create a ServerProxy with a URI that doesn't respond to XMLRPC requests
proxy = xmlrpc.client.ServerProxy("http://google.com/")

try:
    proxy.some_method()
except xmlrpc.client.ProtocolError as err:
    print("A protocol error occurred")
    print("URL: %s" % err.url)
    print("HTTP/HTTPS headers: %s" % err.headers)
    print("Error code: %d" % err.errcode)
    print("Error message: %s" % err.errmsg)
```

22.26.6 Objetos MultiCall

The *MultiCall* object provides a way to encapsulate multiple calls to a remote server into a single request[1].

class xmlrpc.client.**MultiCall**(*server*)

Create an object used to boxcar method calls. *server* is the eventual target of the call. Calls can be made to the result object, but they will immediately return `None`, and only store the call name and parameters in the *MultiCall* object. Calling the object itself causes all stored calls to be transmitted as a single `system.multicall` request. The result of this call is a *generator*; iterating over this generator yields the individual results.

[1] This approach has been first presented in a discussion on xmlrpc.com.

A usage example of this class follows. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```

22.26.7 Convenience Functions

xmlrpc.client.**dumps**(*params*, *methodname=None*, *methodresponse=None*, *encoding=None*, *allow_none=False*)

Convert *params* into an XML-RPC request. or into a response if *methodresponse* is true. *params* can be either a tuple of arguments or an instance of the `Fault` exception class. If *methodresponse* is true, only a single value can be returned, meaning that *params* must be of length 1. *encoding*, if supplied, is the encoding to use in the generated XML; the default is UTF-8. Python's `None` value cannot be used in standard XML-RPC; to allow using it via an extension, provide a true value for *allow_none*.

xmlrpc.client.**loads**(*data*, *use_datetime=False*, *use_builtin_types=False*)

Convert an XML-RPC request or response into Python objects, a `(params, methodname)`. *params* is a tuple of argument; *methodname* is a string, or `None` if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a `Fault` exception. The *use_builtin_types* flag can be used to cause date/time values to be presented as `datetime.datetime` objects and binary data to be presented as `bytes` objects; this flag is false by default.

The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

Alterado na versão 3.3: O sinalizador *use_builtin_types* foi adicionado.

22.26.8 Exemplo de uso do cliente

```
# simple test program (from the XML-RPC specification)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # local server
with ServerProxy("http://betty.userland.com") as proxy:

    print(proxy)

    try:
        print(proxy.examples.getStateName(41))
    except Error as v:
        print("ERROR", v)
```

To access an XML-RPC server through a HTTP proxy, you need to define a custom transport. The following example shows how:

```
import http.client
import xmlrpc.client

class ProxiedTransport(xmlrpc.client.Transport):

    def set_proxy(self, host, port=None, headers=None):
        self.proxy = host, port
        self.proxy_headers = headers

    def make_connection(self, host):
        connection = http.client.HTTPConnection(*self.proxy)
        connection.set_tunnel(host, headers=self.proxy_headers)
        self._connection = host, connection
        return connection

transport = ProxiedTransport()
transport.set_proxy('proxy-server', 8080)
server = xmlrpc.client.ServerProxy('http://betty.userland.com', transport=transport)
print(server.examples.getStateName(41))
```

22.26.9 Example of Client and Server Usage

See *Exemplo de SimpleXMLRPCServer*.

22.27 xmlrpc.server — Servidores XML-RPC básicos

Código Fonte: Lib/xmlrpc/server.py

O módulo `xmlrpc.server` fornece um framework básico de servidor para servidores XML-RPC escritos em Python. Os servidores podem ser independentes, usando `SimpleXMLRPCServer`, ou incorporados em um ambiente CGI, usando `CGIXMLRPCRequestHandler`.

> **Aviso:** O módulo `xmlrpc.server` não é seguro contra dados criados com códigos maliciosos. Se você precisar analisar dados não confiáveis ou não autenticados, consulte *XML vulnerabilities*.

class xmlrpc.server.**SimpleXMLRPCServer**(*addr*, *requestHandler=SimpleXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=False*)

Cria uma nova instância do servidor. Esta classe fornece métodos para registro de funções que podem ser chamadas pelo protocolo XML-RPC. O parâmetro *requestHandler* deve ser uma fábrica para instâncias do tratador de solicitações; o padrão é `SimpleXMLRPCRequestHandler`. Os parâmetros *addr* e *requestHandler* são passados para o construtor `socketserver.TCPServer`. Se *logRequests* for true (o padrão), as solicitações serão registradas; definir esse parâmetro como false desativará o log. Os parâmetros *allow_none* e *encoding* são transmitidos para `xmlrpc.client` e controlam as respostas XML-RPC que serão retornadas do servidor. O parâmetro *bind_and_activate* controla se `server_bind()` e `server_activate()` são chamados imediatamente pelo construtor; o padrão é true. A configuração como false permite que o código manipule a variável de classe *allow_reuse_address* antes que o endereço seja vinculado. O parâmetro *use_builtin_types* é passado para a função `loads()` e controla quais tipos são processados quando valores de data/hora ou dados binários são recebidos; o padrão é false.

Alterado na versão 3.3: O sinalizador *use_builtin_types* foi adicionado.

class xmlrpc.server.**CGIXMLRPCRequestHandler**(*allow_none=False*, *encoding=None*, *use_builtin_types=False*)

Cria uma nova instância para manipular solicitações XML-RPC em um ambiente CGI. Os parâmetros *allow_none* e *encoding* são transmitidos para `xmlrpc.client` e controlam as respostas XML-RPC que serão retornadas do servidor. O parâmetro *use_builtin_types* é passado para a função `loads()` e controla quais tipos são processados quando valores de data/hora ou dados binários são recebidos; o padrão é false.

Alterado na versão 3.3: O sinalizador *use_builtin_types* foi adicionado.

class xmlrpc.server.**SimpleXMLRPCRequestHandler**

Cria uma nova instância do manipulador de solicitações. Este manipulador de solicitação possui suporte a solicitações `POST` e modifica o log para que o parâmetro *logRequests* para o construtor de `SimpleXMLRPCServer` seja respeitado.

22.27.1 Objetos de SimpleXMLRPCServer

A classe *SimpleXMLRPCServer* é baseada em *socketserver.TCPServer* e fornece um meio de criar servidores XML-RPC simples e independentes.

SimpleXMLRPCServer.**register_function**(*function=None*, *name=None*)
Registra uma função que possa responder às solicitações XML-RPC. Se *name* for fornecido, será o nome do método associado a *function*, caso contrário, `function.__name__` será usado. *name* é uma string e pode conter caracteres ilegais nos identificadores Python, incluindo o caractere de ponto.

Este método também pode ser usado como um decorador. Quando usado como decorador, *name* só pode ser fornecido como argumento nomeado para registrar *function* em *name*. Se nenhum *nome* for fornecido, `function.__name__` será usado.

Alterado na versão 3.7: *register_function()* pode ser usado como um decorador.

SimpleXMLRPCServer.**register_instance**(*instance*, *allow_dotted_names=False*)
Registra um objeto que é usado para expor nomes de métodos que não foram registrados usando *register_function()*. Se *instance* contiver um método `_dispatch()`, ele será chamado com o nome do método solicitado e os parâmetros da solicitação. Sua API é `def _dispatch(self, method, params)` (observe que *params* não representa uma lista de argumentos variáveis). Se ele chama uma função subjacente para executar sua tarefa, essa função é chamada como `func(*params)`, expandindo a lista de parâmetros. O valor de retorno de `_dispatch()` é retornado ao cliente como resultado. Se *instance* não possui o método `_dispatch()`, é procurado por um atributo correspondente ao nome do método solicitado.

Se o argumento opcional *allow_dotted_names* for true e a instância não tiver o método `_dispatch()`, se o nome do método solicitado contiver pontos, cada componente do nome do método será pesquisado individualmente, com o efeito de que um simples pesquisa hierárquica é realizada. O valor encontrado nessa pesquisa é chamado com os parâmetros da solicitação e o valor retornado é passado de volta ao cliente.

> **Aviso:** A ativação da opção *allow_dotted_names* permite que os invasores acessem as variáveis globais do seu módulo e podem permitir que os invasores executem códigos arbitrários em sua máquina. Use esta opção apenas em uma rede fechada e segura.

SimpleXMLRPCServer.**register_introspection_functions**()
Registradores de funções de introspecção XML-RPC `system.listMethods`, `system.methodHelp` e `system.methodSignature`.

SimpleXMLRPCServer.**register_multicall_functions**()
Registra a função de multichamada XML-RPC system.multicall.

SimpleXMLRPCRequestHandler.**rpc_paths**
Um valor de atributo que deve ser uma tupla listando partes do caminho válidas da URL para receber solicitações XML-RPC. Solicitações postadas em outros caminhos resultarão em um erro HTTP 404 “página inexistente”. Se esta tupla estiver vazia, todos os caminhos serão considerados válidos. O valor padrão é `('/', '/RPC2')`.

Exemplo de SimpleXMLRPCServer

Código do servidor:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name
    def adder_function(x, y):
        return x + y
    server.register_function(adder_function, 'add')

    # Register an instance; all the methods of the instance are
    # published as XML-RPC methods (in this case, just 'mul').
    class MyFuncs:
        def mul(self, x, y):
            return x * y

    server.register_instance(MyFuncs())

    # Run the server's main loop
    server.serve_forever()
```

O código do cliente a seguir chamará os métodos disponibilizados pelo servidor anterior:

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3))  # Returns 2**3 = 8
print(s.add(2,3))  # Returns 5
print(s.mul(5,2))  # Returns 5*2 = 10

# Print list of available methods
print(s.system.listMethods())
```

`register_function()` também pode ser usado como um decorador. O exemplo anterior do servidor pode registrar funções de um jeito decorador:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

with SimpleXMLRPCServer(('localhost', 8000),
```

(continua na próxima página)

(continuação da página anterior)

```
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name, using
    # register_function as a decorator. *name* can only be given
    # as a keyword argument.
    @server.register_function(name='add')
    def adder_function(x, y):
        return x + y

    # Register a function under function.__name__.
    @server.register_function
    def mul(x, y):
        return x * y

    server.serve_forever()
```

O exemplo a seguir incluído no módulo `Lib/xmlrpc/server.py` mostra um servidor que permite nomes com pontos e registra uma função de várias chamadas.

Aviso: A ativação da opção *allow_dotted_names* permite que os invasores acessem as variáveis globais do seu módulo e podem permitir que os invasores executem códigos arbitrários em sua máquina. Use este exemplo apenas em uma rede fechada e segura.

```
import datetime

class ExampleService:
    def getData(self):
        return '42'

    class currentTime:
        @staticmethod
        def getCurrentTime():
            return datetime.datetime.now()

with SimpleXMLRPCServer(("localhost", 8000)) as server:
    server.register_function(pow)
    server.register_function(lambda x,y: x+y, 'add')
    server.register_instance(ExampleService(), allow_dotted_names=True)
    server.register_multicall_functions()
    print('Serving XML-RPC on localhost port 8000')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\nKeyboard interrupt received, exiting.")
        sys.exit(0)
```

Esta demonstração ExampleService pode ser chamada na linha de comando:

```
python -m xmlrpc.server
```

O cliente que interage com o servidor acima está incluído em *Lib/xmlrpc/client.py*:

```
server = ServerProxy("http://localhost:8000")

try:
    print(server.currentTime.getCurrentTime())
except Error as v:
    print("ERROR", v)

multi = MultiCall(server)
multi.getData()
multi.pow(2,9)
multi.add(1,2)
try:
    for response in multi():
        print(response)
except Error as v:
    print("ERROR", v)
```

Este cliente que interage com o servidor XMLRPC de demonstração pode ser chamado como:

```
python -m xmlrpc.client
```

22.27.2 CGIXMLRPCRequestHandler

A classe *CGIXMLRPCRequestHandler* pode ser usada para lidar com solicitações XML-RPC enviadas para scripts CGI do Python.

CGIXMLRPCRequestHandler.**register_function**(*function=None*, *name=None*)
Registra uma função que possa responder às solicitações XML-RPC. Se *name* for fornecido, será o nome do método associado a *function*, caso contrário, `function.__name__` será usado. *name* é uma string e pode conter caracteres ilegais nos identificadores Python, incluindo o caractere de ponto.

Este método também pode ser usado como um decorador. Quando usado como decorador, *name* só pode ser fornecido como argumento nomeado para registrar *function* em *name*. Se nenhum *nome* for fornecido, `function.__name__` será usado.

Alterado na versão 3.7: *register_function()* pode ser usado como um decorador.

CGIXMLRPCRequestHandler.**register_instance**(*instance*)
Registra um objeto que é usado para expor nomes de métodos que não foram registrados usando *register_function()*. Se a instância contiver um método `_dispatch()`, ela será chamada com o nome do método solicitado e os parâmetros da solicitação; o valor retornado é retornado ao cliente como resultado. Se a instância não tiver um método `_dispatch()`, será procurado um atributo correspondente ao nome do método solicitado; se o nome do método solicitado contiver pontos, cada componente do nome do método será pesquisado individualmente, com o efeito de que uma pesquisa hierárquica simples é executada. O valor encontrado nessa pesquisa é chamado com os parâmetros da solicitação e o valor retornado é passado de volta ao cliente.

CGIXMLRPCRequestHandler.**register_introspection_functions**()
Registra as funções de introspecção XML-RPC `system.listMethods`, `system.methodHelp` e `system.methodSignature`.

CGIXMLRPCRequestHandler.**register_multicall_functions**()
Registra a função de multichamada XML-RPC `system.multicall`.

CGIXMLRPCRequestHandler.**handle_request**(*request_text=None*)
Manipula uma solicitação XML-RPC. Se *request_text* for fornecido, devem ser os dados POST fornecidos pelo servidor HTTP, caso contrário, o conteúdo do stdin será usado.

Exemplo:

```
class MyFuncs:
    def mul(self, x, y):
        return x * y


handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

22.27.3 Documentando servidor XMLRPC

Essas classes estendem as classes acima para servir a documentação HTML em resposta a solicitações HTTP GET. Os servidores podem ser independentes, usando *DocXMLRPCServer* ou incorporados em um ambiente CGI, usando *DocCGIXMLRPCRequestHandler*.

class xmlrpc.server.**DocXMLRPCServer**(*addr*, *requestHandler=DocXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=True*)

Cria uma nova instância do servidor. Todos os parâmetros têm o mesmo significado que para *SimpleXMLRPCServer*; *requestHandler* assume como padrão *DocXMLRPCRequestHandler*.

Alterado na versão 3.3: O sinalizador *use_builtin_types* foi adicionado.

class xmlrpc.server.**DocCGIXMLRPCRequestHandler**

Cria uma nova instância para manipular solicitações XML-RPC em um ambiente CGI.

class xmlrpc.server.**DocXMLRPCRequestHandler**

Cria uma nova instância do manipulador de solicitações. Este manipulador de solicitações possui suporte a solicitações POST de XML-RPC, documenta solicitações GET e modifica o log para que o parâmetro *logRequests* no parâmetro *DocXMLRPCServer* seja respeitado.

22.27.4 Objetos de DocXMLRPCServer

A classe *DocXMLRPCServer* é derivada de *SimpleXMLRPCServer* e fornece um meio de criar servidores XML-RPC autodocumentados e independentes. Solicitações HTTP POST são tratadas como chamadas de método XML-RPC. As solicitações HTTP GET são tratadas gerando documentação HTML no estilo pydoc. Isso permite que um servidor forneça sua própria documentação baseada na Web.

DocXMLRPCServer.**set_server_title**(*server_title*)

Define o título usado na documentação HTML gerada. Este título será usado dentro do elemento "title" do HTML.

DocXMLRPCServer.**set_server_name**(*server_name*)

Define o nome usado na documentação HTML gerada. Este nome aparecerá na parte superior da documentação gerada dentro de um elemento "h1".

DocXMLRPCServer.**set_server_documentation**(*server_documentation*)

Define a descrição usada na documentação HTML gerada. Esta descrição aparecerá como um parágrafo, abaixo do nome do servidor, na documentação.

22.27.5 DocCGIXMLRPCRequestHandler

A classe *DocCGIXMLRPCRequestHandler* é derivada de *CGIXMLRPCRequestHandler* e fornece um meio de criar scripts CGI XML-RPC autodocumentados. Solicitações HTTP POST são tratadas como chamadas de método XML-RPC. As solicitações HTTP GET são tratadas gerando documentação HTML no estilo pydoc. Isso permite que um servidor forneça sua própria documentação baseada na web.

DocCGIXMLRPCRequestHandler.**set_server_title**(*server_title*)
Define o título usado na documentação HTML gerada. Este título será usado dentro do elemento "title" do HTML.

DocCGIXMLRPCRequestHandler.**set_server_name**(*server_name*)
Define o nome usado na documentação HTML gerada. Este nome aparecerá na parte superior da documentação gerada dentro de um elemento "h1".

DocCGIXMLRPCRequestHandler.**set_server_documentation**(*server_documentation*)
Define a descrição usada na documentação HTML gerada. Esta descrição aparecerá como um parágrafo, abaixo do nome do servidor, na documentação.

22.28 ipaddress — IPv4/IPv6 manipulation library

Código Fonte: Lib/ipaddress.py

---

ipaddress provides the capabilities to create, manipulate and operate on IPv4 and IPv6 addresses and networks.

The functions and classes in this module make it straightforward to handle various tasks related to IP addresses, including checking whether or not two hosts are on the same subnet, iterating over all hosts in a particular subnet, checking whether or not a string represents a valid IP address or network definition, and so on.

This is the full module API reference—for an overview and introduction, see ipaddress-howto.

Novo na versão 3.3.

22.28.1 Convenience factory functions

The *ipaddress* module provides factory functions to conveniently create IP addresses, networks and interfaces:

ipaddress.**ip_address**(*address*)
Return an *IPv4Address* or *IPv6Address* object depending on the IP address passed as argument. Either IPv4 or IPv6 addresses may be supplied; integers less than 2**32 will be considered to be IPv4 by default. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address.

```
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')
```

ipaddress.**ip_network**(*address*, *strict=True*)
Return an *IPv4Network* or *IPv6Network* object depending on the IP address passed as argument. *address* is a string or integer representing the IP network. Either IPv4 or IPv6 networks may be supplied; integers less than 2**32 will be considered to be IPv4 by default. *strict* is passed to *IPv4Network* or *IPv6Network* constructor. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address, or if the network has host bits set.

```
>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')
```

ipaddress.**ip_interface**(*address*)
Return an *IPv4Interface* or *IPv6Interface* object depending on the IP address passed as argument. *address* is a string or integer representing the IP address. Either IPv4 or IPv6 addresses may be supplied; integers less than 2**32 will be considered to be IPv4 by default. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address.

One downside of these convenience functions is that the need to handle both IPv4 and IPv6 formats means that error messages provide minimal information on the precise error, as the functions don't know whether the IPv4 or IPv6 format was intended. More detailed error reporting can be obtained by calling the appropriate version specific class constructors directly.

22.28.2 IP Addresses

Endereço de objetos

The *IPv4Address* and *IPv6Address* objects share a lot of common attributes. Some attributes that are only meaningful for IPv6 addresses are also implemented by *IPv4Address* objects, in order to make it easier to write code that handles both IP versions correctly. Address objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.**IPv4Address**(*address*)
Construct an IPv4 address. An *AddressValueError* is raised if *address* is not a valid IPv4 address.

O seguinte constitui um endereço IPv4 válido:

1. Uma string em notação decimal por ponto, consistindo de quatro inteiros decimais em um intervalo inclusivo 0–255 separado por pontos (e.g. `192.168.0.1`). Cada inteiro representa um octeto (byte) no endereço. Zeros à esquerda são tolerados apenas para valores menores que 8 (por não haver ambiguidade entre a interpretação de decimal ou octal para tais strings).
2. Um inteiro que cabe em 32 bits.
3. An integer packed into a *bytes* object of length 4 (most significant octet first).

```
>>> ipaddress.IPv4Address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(3232235521)
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(b'\xC0\xA8\x00\x01')
IPv4Address('192.168.0.1')
```

version
The appropriate version number: `4` for IPv4, `6` for IPv6.

max_prefixlen
The total number of bits in the address representation for this version: `32` for IPv4, `128` for IPv6.

The prefix defines the number of leading bits in an address that are compared to determine whether or not an address is part of a network.

compressed

exploded
The string representation in dotted decimal notation. Leading zeroes are never included in the representation.

As IPv4 does not define a shorthand notation for addresses with octets set to zero, these two attributes are always the same as `str(addr)` for IPv4 addresses. Exposing these attributes makes it easier to write display code that can handle both IPv4 and IPv6 addresses.

packed
The binary representation of this address - a *bytes* object of the appropriate length (most significant octet first). This is 4 bytes for IPv4 and 16 bytes for IPv6.

reverse_pointer
The name of the reverse DNS PTR record for the IP address, e.g.:

```
>>> ipaddress.ip_address("127.0.0.1").reverse_pointer
'1.0.0.127.in-addr.arpa'
>>> ipaddress.ip_address("2001:db8::1").reverse_pointer
'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa'
```

This is the name that could be used for performing a PTR lookup, not the resolved hostname itself.

Novo na versão 3.5.

is_multicast
`True` if the address is reserved for multicast use. See **RFC 3171** (for IPv4) or **RFC 2373** (for IPv6).

is_private
`True` if the address is allocated for private networks. See iana-ipv4-special-registry (for IPv4) or iana-ipv6-special-registry (for IPv6).

is_global
`True` if the address is allocated for public networks. See iana-ipv4-special-registry (for IPv4) or iana-ipv6-special-registry (for IPv6).

Novo na versão 3.4.

is_unspecified
`True` if the address is unspecified. See **RFC 5735** (for IPv4) or **RFC 2373** (for IPv6).

is_reserved
`True` if the address is otherwise IETF reserved.

is_loopback
`True` if this is a loopback address. See **RFC 3330** (for IPv4) or **RFC 2373** (for IPv6).

is_link_local
`True` se o endereço está reservado para uso de link local. Ver: **RFC 3927**.

class ipaddress.**IPv6Address**(*address*)
Construct an IPv6 address. An *AddressValueError* is raised if *address* is not a valid IPv6 address.

O seguinte constitui um endereço IPv6 válido:

1. Uma string constituída de oito grupos de quatro dígitos hexadecimais, cada grupo representando 16 bits. Os grupos são separados por dois pontos. Isto descreve uma notação *explodida* (longa); A string também pode ser *compactada* (notação curta) por vários meios. Ver **RFC 4291** para detalhes. Por exemplo, `"0000:0000:0000:0000:0000:0abc:0007:0def"` pode ser compactada para `"::abc:7:def"`.
2. Um inteiro que cabe em 128 bits.
3. An integer packed into a *bytes* object of length 16, big-endian.

```
>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')
```

compressed

A forma curta da representação do endereço, com zeros à esquerda em grupos omitidos e a sequência mais longa de grupos consistida inteiramente por zeros colapsada em um grupo vazio único.

Este também é o valor retornado por `str(addr)` para endereços IPv6.

exploded

A forma longa da representação do endereço, com todos zeros à esquerda em grupos consistindo inteiramente de zeros incluídos.

For the following attributes, see the corresponding documentation of the *`IPv4Address`* class:

packed

reverse_pointer

version

max_prefixlen

is_multicast

is_private

is_global

is_unspecified

is_reserved

is_loopback

is_link_local
Novo na versão 3.4: is_global

is_site_local
`True` if the address is reserved for site-local usage. Note that the site-local address space has been deprecated by **RFC 3879**. Use *`is_private`* to test if this address is in the space of unique local addresses as defined by **RFC 4193**.

ipv4_mapped
For addresses that appear to be IPv4 mapped addresses (starting with `::FFFF/96`), this property will report the embedded IPv4 address. For any other address, this property will be `None`.

sixtofour
For addresses that appear to be 6to4 addresses (starting with `2002::/16`) as defined by **RFC 3056**, this property will report the embedded IPv4 address. For any other address, this property will be `None`.

teredo
For addresses that appear to be Teredo addresses (starting with `2001::/32`) as defined by **RFC 4380**, this property will report the embedded `(server, client)` IP address pair. For any other address, this property will be `None`.

Conversion to Strings and Integers

To interoperate with networking interfaces such as the socket module, addresses must be converted to strings or integers. This is handled using the *str()* and *int()* builtin functions:

```
>>> str(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> int(ipaddress.IPv4Address('192.168.0.1'))
3232235521
>>> str(ipaddress.IPv6Address('::1'))
'::1'
>>> int(ipaddress.IPv6Address('::1'))
1
```

Operadores

Address objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Comparison operators

Address objects can be compared with the usual set of comparison operators. Some examples:

```
>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True
```

Operadores aritméticos

Integers can be added to or subtracted from address objects. Some examples:

```
>>> IPv4Address('127.0.0.2') + 3
IPv4Address('127.0.0.5')
>>> IPv4Address('127.0.0.2') - 3
IPv4Address('126.255.255.255')
>>> IPv4Address('255.255.255.255') + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ipaddress.AddressValueError: 4294967296 (>= 2**32) is not permitted as an IPv4 address
```

22.28.3 IP Network definitions

The *IPv4Network* and *IPv6Network* objects provide a mechanism for defining and inspecting IP network definitions. A network definition consists of a *mask* and a *network address*, and as such defines a range of IP addresses that equal the network address when masked (binary AND) with the mask. For example, a network definition with the mask `255.255.255.0` and the network address `192.168.1.0` consists of IP addresses in the inclusive range `192.168.1.0` to `192.168.1.255`.

Prefix, net mask and host mask

There are several equivalent ways to specify IP network masks. A *prefix* `/<nbits>` is a notation that denotes how many high-order bits are set in the network mask. A *net mask* is an IP address with some number of high-order bits set. Thus the prefix `/24` is equivalent to the net mask `255.255.255.0` in IPv4, or `ffff:ff00::` in IPv6. In addition, a *host mask* is the logical inverse of a *net mask*, and is sometimes used (for example in Cisco access control lists) to denote a network mask. The host mask equivalent to `/24` in IPv4 is `0.0.0.255`.

Objetos Network

All attributes implemented by address objects are implemented by network objects as well. In addition, network objects implement additional attributes. All of these are common between *IPv4Network* and *IPv6Network*, so to avoid duplication they are only documented for *IPv4Network*. Network objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.**IPv4Network**(*address*, *strict=True*)

Construct an IPv4 network definition. *address* can be one of the following:

1. A string consisting of an IP address and an optional mask, separated by a slash (`/`). The IP address is the network address, and the mask can be either a single number, which means it's a *prefix*, or a string representation of an IPv4 address. If it's the latter, the mask is interpreted as a *net mask* if it starts with a non-zero field, or as a *host mask* if it starts with a zero field, with the single exception of an all-zero mask which is treated as a *net mask*. If no mask is provided, it's considered to be `/32`.

   For example, the following *address* specifications are equivalent: `192.168.1.0/24`, `192.168.1.0/255.255.255.0` and `192.168.1.0/0.0.0.255`.

2. An integer that fits into 32 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/32`.

3. An integer packed into a *bytes* object of length 4, big-endian. The interpretation is similar to an integer *address*.

4. A two-tuple of an address description and a netmask, where the address description is either a string, a 32-bits integer, a 4-bytes packed integer, or an existing IPv4Address object; and the netmask is either an integer representing the prefix length (e.g. `24`) or a string representing the prefix mask (e.g. `255.255.255.0`).

An *AddressValueError* is raised if *address* is not a valid IPv4 address. A *NetmaskValueError* is raised if the mask is not valid for an IPv4 address.

If *strict* is `True` and host bits are set in the supplied address, then *ValueError* is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Unless stated otherwise, all network methods accepting other network/address objects will raise *TypeError* if the argument's IP version is incompatible to `self`.

Alterado na versão 3.5: Added the two-tuple form for the *address* constructor parameter.

version

max_prefixlen
Refer to the corresponding attribute documentation in *IPv4Address*.

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local
These attributes are true for the network as a whole if they are true for both the network address and the broadcast address.

network_address
The network address for the network. The network address and the prefix length together uniquely define a network.

broadcast_address
The broadcast address for the network. Packets sent to the broadcast address should be received by every host on the network.

hostmask
The host mask, as an *IPv4Address* object.

netmask
The net mask, as an *IPv4Address* object.

with_prefixlen

compressed

exploded
A string representation of the network, with the mask in prefix notation.

`with_prefixlen` and `compressed` are always the same as `str(network)`. `exploded` uses the exploded form the network address.

with_netmask
A string representation of the network, with the mask in net mask notation.

with_hostmask
A string representation of the network, with the mask in host mask notation.

num_addresses
O número total de endereços na rede.

prefixlen
Comprimento do prefixo de rede, em bits.

hosts()
Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the network address itself and the network broadcast address. For networks with a mask length of 31, the network address and network broadcast address are also included in the result.

```
>>> list(ip_network('192.0.2.0/29').hosts())
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'),
 IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'),
 IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]
>>> list(ip_network('192.0.2.0/31').hosts())
[IPv4Address('192.0.2.0'), IPv4Address('192.0.2.1')]
```

overlaps(*other*)
`True` if this network is partly or wholly contained in *other* or *other* is wholly contained in this network.

address_exclude(*network*)
Computes the network definitions resulting from removing the given *network* from this one. Returns an iterator of network objects. Raises `ValueError` if *network* is not completely contained in this network.

```
>>> n1 = ip_network('192.0.2.0/28')
>>> n2 = ip_network('192.0.2.1/32')
>>> list(n1.address_exclude(n2))
[IPv4Network('192.0.2.8/29'), IPv4Network('192.0.2.4/30'),
 IPv4Network('192.0.2.2/31'), IPv4Network('192.0.2.0/32')]
```

subnets(*prefixlen_diff=1*, *new_prefix=None*)
The subnets that join to make the current network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be increased by. *new_prefix* is the desired new prefix of the subnets; it must be larger than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns an iterator of network objects.

```
>>> list(ip_network('192.0.2.0/24').subnets())
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
>>> list(ip_network('192.0.2.0/24').subnets(prefixlen_diff=2))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=26))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=23))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise ValueError('new prefix must be longer')
ValueError: new prefix must be longer
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=25))
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
```

supernet(*prefixlen_diff=1*, *new_prefix=None*)
The supernet containing this network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be decreased by. *new_prefix* is the desired new prefix of the supernet; it must be smaller than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns a single network object.

```
>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.2.0/23')
>>> ip_network('192.0.2.0/24').supernet(prefixlen_diff=2)
IPv4Network('192.0.0.0/22')
>>> ip_network('192.0.2.0/24').supernet(new_prefix=20)
IPv4Network('192.0.0.0/20')
```

subnet_of(*other*)
Retorna `True` se esta rede é uma sub-rede de *outra*.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> b.subnet_of(a)
True
```

Novo na versão 3.7.

supernet_of(*other*)
Retorna `True` se esta rede é uma super-rede de *outra*.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> a.supernet_of(b)
True
```

Novo na versão 3.7.

compare_networks(*other*)
Compare this network to *other*. In this comparison only the network addresses are considered; host bits aren't. Returns either `-1`, `0` or `1`.

```
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.2/32'))
-1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.0/32'))
1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.1/32'))
0
```

Obsoleto desde a versão 3.7: It uses the same ordering and comparison algorithm as "<", "==", and ">"

class ipaddress.**IPv6Network**(*address*, *strict=True*)
Construct an IPv6 network definition. *address* can be one of the following:

1. A string consisting of an IP address and an optional prefix length, separated by a slash (`/`). The IP address is the network address, and the prefix length must be a single number, the *prefix*. If no prefix length is provided, it's considered to be `/128`.

   Note that currently expanded netmasks are not supported. That means `2001:db00::0/24` is a valid argument while `2001:db00::0/ffff:ff00::` not.

2. An integer that fits into 128 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/128`.

3. An integer packed into a *bytes* object of length 16, big-endian. The interpretation is similar to an integer *address*.

4. A two-tuple of an address description and a netmask, where the address description is either a string, a 128-bits integer, a 16-bytes packed integer, or an existing IPv6Address object; and the netmask is an integer representing the prefix length.

An *AddressValueError* is raised if *address* is not a valid IPv6 address. A *NetmaskValueError* is raised if the mask is not valid for an IPv6 address.

If *strict* is `True` and host bits are set in the supplied address, then *ValueError* is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Alterado na versão 3.5: Added the two-tuple form for the *address* constructor parameter.

version

max_prefixlen

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

network_address

broadcast_address

hostmask

netmask

with_prefixlen

compressed

exploded

with_netmask

with_hostmask

num_addresses

prefixlen

hosts()
Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the Subnet-Router anycast address. For networks with a mask length of 127, the Subnet-Router anycast address is also included in the result.

overlaps(*other*)

address_exclude(*network*)

subnets(*prefixlen_diff=1*, *new_prefix=None*)

supernet(*prefixlen_diff=1*, *new_prefix=None*)

subnet_of(*other*)

supernet_of(*other*)

compare_networks(*other*)
Refer to the corresponding attribute documentation in *IPv4Network*.

is_site_local
These attribute is true for the network as a whole if it is true for both the network address and the broadcast address.

Operadores

Network objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Network objects can be compared with the usual set of logical operators. Network objects are ordered first by network address, then by net mask.

Iteração

Network objects can be iterated to list all the addresses belonging to the network. For iteration, *all* hosts are returned, including unusable hosts (for usable hosts, use the `hosts()` method). An example:

```
>>> for addr in IPv4Network('192.0.2.0/28'):
...     addr
...
IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
IPv4Address('192.0.2.13')
IPv4Address('192.0.2.14')
IPv4Address('192.0.2.15')
```

Redes como contêineres de endereços

Network objects can act as containers of addresses. Some examples:

```
>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
>>> IPv4Network('192.0.2.0/28')[15]
IPv4Address('192.0.2.15')
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```

22.28.4 Interface objects

Interface objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.**IPv4Interface**(*address*)

Construct an IPv4 interface. The meaning of *address* is as in the constructor of *IPv4Network*, except that arbitrary host addresses are always accepted.

IPv4Interface is a subclass of *IPv4Address*, so it inherits all the attributes from that class. In addition, the following attributes are available:

ip

The address (*IPv4Address*) without network information.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.ip
IPv4Address('192.0.2.5')
```

network

The network (*IPv4Network*) this interface belongs to.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.network
IPv4Network('192.0.2.0/24')
```

with_prefixlen

A string representation of the interface with the mask in prefix notation.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'
```

with_netmask

A string representation of the interface with the network as a net mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'
```

with_hostmask

A string representation of the interface with the network as a host mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'
```

class ipaddress.**IPv6Interface**(*address*)

Construct an IPv6 interface. The meaning of *address* is as in the constructor of *IPv6Network*, except that arbitrary host addresses are always accepted.

IPv6Interface is a subclass of *IPv6Address*, so it inherits all the attributes from that class. In addition, the following attributes are available:

ip

network

with_prefixlen

with_netmask

with_hostmask
Refer to the corresponding attribute documentation in *IPv4Interface*.

Operadores

Interface objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Interface objects can be compared with the usual set of logical operators.

For equality comparison (== and !=), both the IP address and network must be the same for the objects to be equal. An interface will not compare equal to any address or network object.

For ordering (<, >, etc) the rules are different. Interface and address objects with the same IP version can be compared, and the address objects will always sort before the interface objects. Two interface objects are first compared by their networks and, if those are the same, then by their IP addresses.

22.28.5 Other Module Level Functions

The module also provides the following module level functions:

ipaddress.**v4_int_to_packed**(*address*)
Represent an address as 4 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv4 IP address. A *ValueError* is raised if the integer is negative or too large to be an IPv4 IP address.

```
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.v4_int_to_packed(3221225985)
b'\xc0\x00\x02\x01'
```

ipaddress.**v6_int_to_packed**(*address*)
Represent an address as 16 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv6 IP address. A *ValueError* is raised if the integer is negative or too large to be an IPv6 IP address.

ipaddress.**summarize_address_range**(*first*, *last*)
Return an iterator of the summarized network range given the first and last IP addresses. *first* is the first *IPv4Address* or *IPv6Address* in the range and *last* is the last *IPv4Address* or *IPv6Address* in the range. A *TypeError* is raised if *first* or *last* are not IP addresses or are not of the same version. A *ValueError* is raised if *last* is not greater than *first* or if *first* address version is not 4 or 6.

```
>>> [ipaddr for ipaddr in ipaddress.summarize_address_range(
...    ipaddress.IPv4Address('192.0.2.0'),
...    ipaddress.IPv4Address('192.0.2.130'))]
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'), IPv4Network('192.0.2.
↪130/32')]
```

ipaddress.**collapse_addresses**(*addresses*)
Return an iterator of the collapsed *IPv4Network* or *IPv6Network* objects. *addresses* is an iterator of *IPv4Network* or *IPv6Network* objects. A *TypeError* is raised if *addresses* contains mixed version objects.

```
>>> [ipaddr for ipaddr in
... ipaddress.collapse_addresses([ipaddress.IPv4Network('192.0.2.0/25'),
... ipaddress.IPv4Network('192.0.2.128/25')])]
[IPv4Network('192.0.2.0/24')]
```

ipaddress.**get_mixed_type_key**(*obj*)

Return a key suitable for sorting between networks and addresses. Address and Network objects are not sortable by default; they're fundamentally different, so the expression:

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

doesn't make sense. There are some times however, where you may wish to have *ipaddress* sort these anyway. If you need to do this, you can use this function as the *key* argument to *sorted()*.

obj is either a network or address object.

22.28.6 Custom Exceptions

To support more specific error reporting from class constructors, the module defines the following exceptions:

exception ipaddress.**AddressValueError**(*ValueError*)

Any value error related to the address.

exception ipaddress.**NetmaskValueError**(*ValueError*)

Any value error related to the net mask.

CAPÍTULO 23

Serviços Multimídia

Os módulos descritos neste capítulo implementam vários algoritmos ou interfaces que são bastante úteis em aplicações multimídia. Eles estão disponíveis a critério da instalação. Aqui está uma visão geral:

23.1 `audioop` — Manipulando dados de áudio original

The `audioop` module contains some useful operations on sound fragments. It operates on sound fragments consisting of signed integer samples 8, 16, 24 or 32 bits wide, stored in *bytes-like objects*. All scalar items are integers, unless specified otherwise.

Alterado na versão 3.4: Support for 24-bit samples was added. All functions now accept any *bytes-like object*. String input now results in an immediate error.

This module provides support for a-LAW, u-LAW and Intel/DVI ADPCM encodings.

A few of the more complicated operations only take 16-bit samples, otherwise the sample size (in bytes) is always a parameter of the operation.

The module defines the following variables and functions:

exception audioop.**error**
This exception is raised on all errors, such as unknown number of bytes per sample, etc.

audioop.**add**(*fragment1*, *fragment2*, *width*)
Return a fragment which is the addition of the two samples passed as parameters. *width* is the sample width in bytes, either `1`, `2`, `3` or `4`. Both fragments should have the same length. Samples are truncated in case of overflow.

audioop.**adpcm2lin**(*adpcmfragment*, *width*, *state*)
Decode an Intel/DVI ADPCM coded fragment to a linear fragment. See the description of `lin2adpcm()` for details on ADPCM coding. Return a tuple `(sample, newstate)` where the sample has the width specified in *width*.

audioop.**alaw2lin**(*fragment*, *width*)
Convert sound fragments in a-LAW encoding to linearly encoded sound fragments. a-LAW encoding always uses 8 bits samples, so *width* refers only to the sample width of the output fragment here.

audioop.**avg**(*fragment*, *width*)
Return the average over all samples in the fragment.

audioop.**avgpp**(*fragment*, *width*)
Return the average peak-peak value over all samples in the fragment. No filtering is done, so the usefulness of this routine is questionable.

audioop.**bias**(*fragment*, *width*, *bias*)
Return a fragment that is the original fragment with a bias added to each sample. Samples wrap around in case of overflow.

audioop.**byteswap**(*fragment*, *width*)
"Byteswap" all samples in a fragment and returns the modified fragment. Converts big-endian samples to little-endian and vice versa.

Novo na versão 3.4.

audioop.**cross**(*fragment*, *width*)
Return the number of zero crossings in the fragment passed as an argument.

audioop.**findfactor**(*fragment*, *reference*)
Return a factor *F* such that `rms(add(fragment, mul(reference, -F)))` is minimal, i.e., return the factor with which you should multiply *reference* to make it match as well as possible to *fragment*. The fragments should both contain 2-byte samples.

The time taken by this routine is proportional to `len(fragment)`.

audioop.**findfit**(*fragment*, *reference*)
Try to match *reference* as well as possible to a portion of *fragment* (which should be the longer fragment). This is (conceptually) done by taking slices out of *fragment*, using *findfactor()* to compute the best match, and minimizing the result. The fragments should both contain 2-byte samples. Return a tuple `(offset, factor)` where *offset* is the (integer) offset into *fragment* where the optimal match started and *factor* is the (floating-point) factor as per *findfactor()*.

audioop.**findmax**(*fragment*, *length*)
Search *fragment* for a slice of length *length* samples (not bytes!) with maximum energy, i.e., return *i* for which `rms(fragment[i*2:(i+length)*2])` is maximal. The fragments should both contain 2-byte samples.

The routine takes time proportional to `len(fragment)`.

audioop.**getsample**(*fragment*, *width*, *index*)
Return the value of sample *index* from the fragment.

audioop.**lin2adpcm**(*fragment*, *width*, *state*)
Convert samples to 4 bit Intel/DVI ADPCM encoding. ADPCM coding is an adaptive coding scheme, whereby each 4 bit number is the difference between one sample and the next, divided by a (varying) step. The Intel/DVI ADPCM algorithm has been selected for use by the IMA, so it may well become a standard.

state is a tuple containing the state of the coder. The coder returns a tuple `(adpcmfrag, newstate)`, and the *newstate* should be passed to the next call of *lin2adpcm()*. In the initial call, `None` can be passed as the state. *adpcmfrag* is the ADPCM coded fragment packed 2 4-bit values per byte.

audioop.**lin2alaw**(*fragment*, *width*)
Convert samples in the audio fragment to a-LAW encoding and return this as a bytes object. a-LAW is an audio encoding format whereby you get a dynamic range of about 13 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

audioop.**lin2lin**(*fragment*, *width*, *newwidth*)
Convert samples between 1-, 2-, 3- and 4-byte formats.

Nota: In some audio formats, such as .WAV files, 16, 24 and 32 bit samples are signed, but 8 bit samples are unsigned. So when converting to 8 bit wide samples for these formats, you need to also add 128 to the result:

```
new_frames = audioop.lin2lin(frames, old_width, 1)
new_frames = audioop.bias(new_frames, 1, 128)
```

The same, in reverse, has to be applied when converting from 8 to 16, 24 or 32 bit width samples.

---

audioop.**lin2ulaw**(*fragment*, *width*)
Convert samples in the audio fragment to u-LAW encoding and return this as a bytes object. u-LAW is an audio encoding format whereby you get a dynamic range of about 14 bits using only 8 bit samples. It is used by the Sun audio hardware, among others.

audioop.**max**(*fragment*, *width*)
Return the maximum of the *absolute value* of all samples in a fragment.

audioop.**maxpp**(*fragment*, *width*)
Return the maximum peak-peak value in the sound fragment.

audioop.**minmax**(*fragment*, *width*)
Return a tuple consisting of the minimum and maximum values of all samples in the sound fragment.

audioop.**mul**(*fragment*, *width*, *factor*)
Return a fragment that has all samples in the original fragment multiplied by the floating-point value *factor*. Samples are truncated in case of overflow.

audioop.**ratecv**(*fragment*, *width*, *nchannels*, *inrate*, *outrate*, *state*[, *weightA*[, *weightB*]])
Convert the frame rate of the input fragment.

state is a tuple containing the state of the converter. The converter returns a tuple `(newfragment, newstate)`, and *newstate* should be passed to the next call of *ratecv()*. The initial call should pass `None` as the state.

The *weightA* and *weightB* arguments are parameters for a simple digital filter and default to `1` and `0` respectively.

audioop.**reverse**(*fragment*, *width*)
Reverse the samples in a fragment and returns the modified fragment.

audioop.**rms**(*fragment*, *width*)
Return the root-mean-square of the fragment, i.e. `sqrt(sum(S_i^2)/n)`.

This is a measure of the power in an audio signal.

audioop.**tomono**(*fragment*, *width*, *lfactor*, *rfactor*)
Convert a stereo fragment to a mono fragment. The left channel is multiplied by *lfactor* and the right channel by *rfactor* before adding the two channels to give a mono signal.

audioop.**tostereo**(*fragment*, *width*, *lfactor*, *rfactor*)
Generate a stereo fragment from a mono fragment. Each pair of samples in the stereo fragment are computed from the mono sample, whereby left channel samples are multiplied by *lfactor* and right channel samples by *rfactor*.

audioop.**ulaw2lin**(*fragment*, *width*)
Convert sound fragments in u-LAW encoding to linearly encoded sound fragments. u-LAW encoding always uses 8 bits samples, so *width* refers only to the sample width of the output fragment here.

Note that operations such as *mul()* or *max()* make no distinction between mono and stereo fragments, i.e. all samples are treated equal. If this is a problem the stereo fragment should be split into two mono fragments first and recombined later. Here is an example of how to do that:

```
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
    lsample = audioop.mul(lsample, width, lfactor)
    rsample = audioop.mul(rsample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

If you use the ADPCM coder to build network packets and you want your protocol to be stateless (i.e. to be able to tolerate packet loss) you should not only transmit the data but also the state. Note that you should send the *initial* state (the one you passed to *lin2adpcm()*) along to the decoder, not the final state (as returned by the coder). If you want to use *struct.Struct* to store the state in binary you can code the first element (the predicted value) in 16 bits and the second (the delta index) in 8.

The ADPCM coders have never been tried against other ADPCM coders, only against themselves. It could well be that I misinterpreted the standards in which case they will not be interoperable with the respective standards.

The `find*()` routines might look a bit funny at first sight. They are primarily meant to do echo cancellation. A reasonably fast way to do this is to pick the most energetic piece of the output sample, locate that in the input sample and subtract the whole output sample from the input sample:

```
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)    # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)],
    #              out_test)
    prefill = '\0'*(pos+ipos)*2
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefill + audioop.mul(outputdata, 2, -factor) + postfill
    return audioop.add(inputdata, outputdata, 2)
```

23.2 `aifc` — Lê e escreve arquivos AIFF e AIFC

Código-fonte: Lib/aifc.py

Este módulo fornece suporte para leitura e gravação de arquivos AIFF e AIFF-C. AIFF significa Audio Interchange File Format, um formato para armazenar amostras de áudio digital em um arquivo. AIFF-C é uma versão mais recente do formato que inclui a capacidade de compactar os dados de áudio.

Arquivos de áudio têm vários parâmetros que descrevem os dados de áudio. A taxa de amostragem ou taxa de quadros é o número de vezes por segundo em que o som é amostrado. O número de canais indica se o áudio é mono, estéreo ou quadro. Cada quadro consiste em uma amostra por canal. O tamanho da amostra é o tamanho em bytes de cada amostra. Assim, um quadro consiste em `nchannels * samplesize` bytes e um segundo de áudio consiste em `nchannels * samplesize * framerate` bytes.

Por exemplo, o áudio com qualidade de CD tem um tamanho de amostra de dois bytes (16 bits), usa dois canais (estéreo) e tem uma taxa de quadros de 44.100 quadros/segundo. Isto dá um tamanho de quadro de 4 bytes (2*2), e o valor de um segundo ocupa 2*2*44100 bytes (176.400 bytes).

O módulo *aifc* define a seguinte função:

aifc.**open**(*file*, *mode=None*)
Abre um arquivo AIFF ou AIFF-C e retorna uma instância de objeto com os métodos descritos abaixo. O argumento *file* é uma string nomeando um arquivo ou um objeto de arquivo. *mode* deve ser `'r'` ou `'rb'` quando o arquivo deve ser aberto para leitura, ou `'w'` ou `'wb'` quando o arquivo deve ser aberto para escrita. Se omitido, `file.mode` é usado se existir, caso contrário `'rb'` é usado. Quando usado para escrita, o objeto de arquivo deve ser pesquisável, a menos que você saiba com antecedência quantas amostras você irá escrever no total e usar `writeframesraw()` e `setnframes()`. A função *open()* pode ser usada em um bloco de instrução `with`. Quando o bloco `with` é concluído, o método *close()* é chamado.

Alterado na versão 3.4: Suporte para a instrução `with` foi adicionado.

Objetos retornados por *open()* quando um arquivo é aberto para leitura têm os seguintes métodos:

aifc.**getnchannels**()
Retorna o número de canais de áudio (1 para mono, 2 para estéreo).

aifc.**getsampwidth**()
Retorna o tamanho em bytes de amostras individuais.

aifc.**getframerate**()
Retorna a taxa de amostra (número de quadros de áudio por segundo).

aifc.**getnframes**()
Retorna o número de quadros de áudio no arquivo.

aifc.**getcomptype**()
Retorna um array de bytes de tamanho 4 descrevendo o tipo de compressão usada no arquivo de áudio. Para arquivos AIFF, o valor retornado é `b'NONE'`.

aifc.**getcompname**()
Retorna um array de bytes convertível para uma descrição legível por humanos do tipo de compactação usado no arquivo de áudio. Para arquivos AIFF, o valor retornado é `b'not compressed'`.

aifc.**getparams**()
Retorna a *namedtuple()* `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, equivalente à saída dos métodos `get*()`.

aifc.**getmarkers**()
Retorna uma lista de marcadores no arquivo de áudio. Um marcador consiste em uma tupla de três elementos. O primeiro é o ID da marca (um inteiro), o segundo é a posição da marca nos quadros desde o início dos dados (um inteiro), o terceiro é o nome da marca (uma string).

aifc.**getmark**(*id*)
Retorna a tupla como descrito em *getmarkers()* para a marca com o *id* fornecido.

aifc.**readframes**(*nframes*)
Lê e retorna os próximos *nframes* quadros do arquivo de áudio. Os dados retornados são uma string contendo para cada quadro as amostras descompactadas de todos os canais.

aifc.**rewind**()
Reinicia o ponteiro de leitura. O próximo *readframes()* começará do início.

aifc.**setpos**(*pos*)
Procura o número do quadro especificado.

aifc.**tell**()
Retorna o número do quadro atual.

aifc.**close**()
Fecha o arquivo AIFF. Depois de chamar esse método, o objeto não pode mais ser usado.

Objetos retornados por *open()* quando um arquivo é aberto para escrita possuem todos os métodos acima, exceto `readframes()` e `setpos()`. Além disso, os seguintes métodos existem. Os métodos `get*()` só podem ser chamados após os métodos `set*()` correspondentes terem sido chamados. Antes do primeiro `writeframes()` ou `writeframesraw()`, todos os parâmetros, exceto o número de quadros, devem ser preenchidos.

aifc.**aiff**()
Cria um arquivo AIFF. O padrão é que um arquivo AIFF-C seja criado, a menos que o nome do arquivo termine em `'.aiff'`, caso em que o padrão é um arquivo AIFF.

aifc.**aifc**()
Cria um arquivo AIFF-C. O padrão é que um arquivo AIFF-C seja criado, a menos que o nome do arquivo termine em `'.aiff'`, caso em que o padrão é um arquivo AIFF.

aifc.**setnchannels**(*nchannels*)
Especifica o número de canais no arquivo de áudio.

aifc.**setsampwidth**(*width*)
Especifica o tamanho em bytes de amostras de áudio.

aifc.**setframerate**(*rate*)
Especifica a frequência de amostragem em quadros por segundo.

aifc.**setnframes**(*nframes*)
Especifica o número de quadros que devem ser escritos no arquivo de áudio. Se este parâmetro não estiver configurado ou estiver incorretamente configurado, o arquivo precisará ter suporte a procura.

aifc.**setcomptype**(*type*, *name*)
Especifica o tipo de compactação. Se não for especificado, os dados de áudio não serão compactados. Em arquivos AIFF, a compactação não é possível. O parâmetro de nome deve ser uma descrição legível por humanos do tipo de compressão como uma array de bytes, o parâmetro de tipo deve ser uma array de bytes de tamanho 4. Atualmente, há suporte aos seguintes tipos de compactação: `b'NONE'`, `b'ULAW'`, `b'ALAW'`, `b'G722'`.

aifc.**setparams**(*nchannels*, *sampwidth*, *framerate*, *comptype*, *compname*)
Define todos os parâmetros acima de uma só vez. O argumento é uma tupla que consiste em vários parâmetros. Isto significa que é possível usar o resultado de uma chamada de *getparams()* como argumento para *setparams()*.

aifc.**setmark**(*id*, *pos*, *name*)
Adiciona uma marca com o ID fornecido (maior que 0) e o nome dado na posição determinada. Este método pode ser chamado a qualquer momento antes de *close()*.

aifc.**tell**()
Retorna a posição atual de escrita no arquivo de saída. Útil em combinação com *setmark()*.

aifc.**writeframes**(*data*)
Escreve dados no arquivo de saída. Este método só pode ser chamado após os parâmetros do arquivo de áudio terem sido definidos.

Alterado na versão 3.4: Todo objeto do tipo byte agora é aceito.

aifc.**writeframesraw**(*data*)
Semelhante a *writeframes()*, exceto que o cabeçalho do arquivo de áudio não é atualizado.

Alterado na versão 3.4: Todo objeto do tipo byte agora é aceito.

aifc.**close**()
Fecha o arquivo AIFF. O cabeçalho do arquivo é atualizado para refletir o tamanho real dos dados de áudio. Depois de chamar esse método, o objeto não pode mais ser usado.

23.3 sunau — Lê e escreve arquivos AU da Sun

Código Fonte: Lib/sunau.py

O módulo *sunau* fornece uma interface conveniente para o formato de som AU da Sun. Observe que este módulo é compatível com a interface dos módulos *aifc* e *wave*.

Um arquivo de áudio consiste em um cabeçalho seguido pelos dados. Os campos do cabeçalho são:

Campo	Conteúdo
palavra mágica	O `.snd` da quatro bytes.
tamanho do cabeçalho	Tamanho do cabeçalho, incluindo informações, em bytes.
tamanho dos dados	Tamanho físico dos dados, em bytes.
codificação	Indica como as amostras de áudio estão codificadas.
taxa de amostra	A taxa de amostra.
nº de canais	O número de canais nas amostras.
informações	Uma string ASCII dando uma descrição do arquivo de áudio (preenchendo com bytes nulos).

Além do campo de informações, todos os campos de cabeçalho têm 4 bytes de tamanho. Eles são todos inteiros sem sinal de 32 bits, codificados na ordem de bytes big-endian.

O módulo *sunau* define as seguintes funções:

sunau.**open**(*file*, *mode*)
Se *file* for uma string, abra o arquivo com esse nome; caso contrário, trata-o como um objeto arquivo ou similar que pode ser procurado. *mode* pode ser qualquer um entre

'r' Modo somente para leitura

'w' Apenas modo de escrita.

Observe que ele não permite arquivos de leitura e escrita.

Um *mode* de `'r'` retorna um objeto `AU_read`, enquanto um *mode* de `'w'` ou `'wb'` retorna um objeto `AU_write`.

sunau.**openfp**(*file*, *mode*)
A synonym for *open()*, maintained for backwards compatibility.

Deprecated since version 3.7, will be removed in version 3.9.

O módulo *sunau* define a seguinte exceção:

exception sunau.**Error**
Um erro levantado quando algo é impossível devido às especificações AU da Sun ou deficiência de implementação.

O módulo *sunau* define os seguintes itens de dados:

sunau.**AUDIO_FILE_MAGIC**
Um número inteiro com o qual todo arquivo AU da Sun válido começa, armazenado no formato big-endian. Esta é a string `.snd` interpretada como um inteiro.

sunau.**AUDIO_FILE_ENCODING_MULAW_8**

sunau.**AUDIO_FILE_ENCODING_LINEAR_8**
sunau.**AUDIO_FILE_ENCODING_LINEAR_16**
sunau.**AUDIO_FILE_ENCODING_LINEAR_24**
sunau.**AUDIO_FILE_ENCODING_LINEAR_32**
sunau.**AUDIO_FILE_ENCODING_ALAW_8**
Valores do campo de codificação do cabeçalho de AU que são suportados por este módulo.

sunau.**AUDIO_FILE_ENCODING_FLOAT**
sunau.**AUDIO_FILE_ENCODING_DOUBLE**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G721**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G722**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G723_3**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G723_5**
Valores conhecidos adicionais de codificação do cabeçalho de AU, mas que não são suportados por este módulo.

23.3.1 Objetos AU_read

Objetos AU_read, conforme retornado por *open()* acima, têm os seguintes métodos:

AU_read.**close**()
Fecha o fluxo e torna a instância inutilizável. (Isso é chamado automaticamente na exclusão.)

AU_read.**getnchannels**()
Retorna o número de canais de áudio (1 para mono, 2 para estéreo).

AU_read.**getsampwidth**()
Retorna a largura da amostra em bytes.

AU_read.**getframerate**()
Retorna a frequência de amostragem.

AU_read.**getnframes**()
Retorna o número de quadros de áudio.

AU_read.**getcomptype**()
Retorna o tipo de compressão. Os tipos de compressão suportados são `'ULAW'`, `'ALAW'` e `'NONE'`.

AU_read.**getcompname**()
Versão legível de *getcomptype()*. Os tipos suportados têm os respectivos nomes `'CCITT G.711 u-law'`, `'CCITT G.711 A-law'` e `'not compressed'`.

AU_read.**getparams**()
Retorna a *namedtuple()* `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, equivalente à saída dos métodos `get*()`.

AU_read.**readframes**(*n*)
Lê e retorna no máximo *n* quadros de áudio, como um objeto *bytes*. Os dados serão retornados em formato linear. Se os dados originais estiverem no formato u-LAW, eles serão convertidos.

AU_read.**rewind**()
Retrocede o ponteiro do arquivo para o início do fluxo de áudio.

Os dois métodos a seguir definem um termo "posição" que é compatível entre eles e é dependente da implementação.

AU_read.**setpos**(*pos*)
Define o ponteiro do arquivo para a posição especificada. Apenas os valores retornados de *tell()* devem ser usados para *pos*.

AU_read.**tell**()
Retorna a posição atual do ponteiro do arquivo. Observe que o valor retornado não tem nada a ver com a posição real no arquivo.

As duas funções a seguir são definidas para compatibilidade com o *aifc*, e não fazem nada de interessante.

AU_read.**getmarkers**()
Retorna None.

AU_read.**getmark**(*id*)
Levanta um erro.

23.3.2 Objetos AU_write

Objetos AU_write, conforme retornado por *open()* acima, têm os seguintes métodos:

AU_write.**setnchannels**(*n*)
Define o número de canais.

AU_write.**setsampwidth**(*n*)
Define a largura da amostra (em bytes).

Alterado na versão 3.4: Adicionado suporte para amostras de 24 bits.

AU_write.**setframerate**(*n*)
Define a taxa de quadros.

AU_write.**setnframes**(*n*)
Define o número de quadros. Isso pode ser alterado posteriormente, quando e se mais quadros forem gravados.

AU_write.**setcomptype**(*type*, *name*)
Define o tipo de compactação e a descrição. Somente 'NONE' e 'ULAW' são suportados na saída.

AU_write.**setparams**(*tuple*)
A *tuple* deve ser (nchannels, sampwidth, framerate, nframes, comptype, compname), com valores válidos para os métodos set*(). Define todos os parâmetros.

AU_write.**tell**()
Retorna a posição atual no arquivo, com as mesmas observações dos métodos *AU_read.tell()* e *AU_read.setpos()*.

AU_write.**writeframesraw**(*data*)
Escreve quadros de áudio, sem corrigir *nframes*.

Alterado na versão 3.4: Todo objeto do tipo byte agora é aceito.

AU_write.**writeframes**(*data*)
Escreve quadros de áudio e certifica-se de que *nframes* esteja correto.

Alterado na versão 3.4: Todo objeto do tipo byte agora é aceito.

AU_write.**close**()
Certifica-se de que *nframes* está correto e fecha o arquivo.

Este método é chamado após a exclusão.

Note that it is invalid to set any parameters after calling writeframes() or writeframesraw().

23.4 wave — Read and write WAV files

Código Fonte: Lib/wave.py

The *wave* module provides a convenient interface to the WAV sound format. It does not support compression/decompression, but it does support mono/stereo.

The *wave* module defines the following function and exception:

wave.**open**(*file*, *mode=None*)
If *file* is a string, open the file by that name, otherwise treat it as a file-like object. *mode* can be:

'rb' Modo somente para leitura

'wb' Apenas modo de escrita.

Note that it does not allow read/write WAV files.

A *mode* of 'rb' returns a Wave_read object, while a *mode* of 'wb' returns a Wave_write object. If *mode* is omitted and a file-like object is passed as *file*, file.mode is used as the default value for *mode*.

If you pass in a file-like object, the wave object will not close it when its close() method is called; it is the caller's responsibility to close the file object.

The *open()* function may be used in a with statement. When the with block completes, the *Wave_read.close()* or *Wave_write.close()* method is called.

Alterado na versão 3.4: Added support for unseekable files.

wave.**openfp**(*file*, *mode*)
A synonym for *open()*, maintained for backwards compatibility.

Deprecated since version 3.7, will be removed in version 3.9.

exception wave.**Error**
An error raised when something is impossible because it violates the WAV specification or hits an implementation deficiency.

23.4.1 Objetos Wave_read

Wave_read objects, as returned by *open()*, have the following methods:

Wave_read.**close**()
Close the stream if it was opened by *wave*, and make the instance unusable. This is called automatically on object collection.

Wave_read.**getnchannels**()
Returns number of audio channels (1 for mono, 2 for stereo).

Wave_read.**getsampwidth**()
Retorna a largura da amostra em bytes.

Wave_read.**getframerate**()
Retorna a frequência de amostragem.

Wave_read.**getnframes**()
Retorna o número de quadros de áudio.

Wave_read.**getcomptype**()
Returns compression type ('NONE' is the only supported type).

Wave_read.**getcompname**()
Human-readable version of *getcomptype()*. Usually 'not compressed' parallels 'NONE'.

Wave_read.**getparams**()
Retorna a *namedtuple()* (nchannels, sampwidth, framerate, nframes, comptype, compname), equivalente à saída dos métodos get*().

Wave_read.**readframes**(*n*)
Reads and returns at most *n* frames of audio, as a *bytes* object.

Wave_read.**rewind**()
Retrocede o ponteiro do arquivo para o início do fluxo de áudio.

The following two methods are defined for compatibility with the *aifc* module, and don't do anything interesting.

Wave_read.**getmarkers**()
Retorna None.

Wave_read.**getmark**(*id*)
Levanta um erro.

Os dois métodos a seguir definem um termo "posição" que é compatível entre eles e é dependente da implementação.

Wave_read.**setpos**(*pos*)
Set the file pointer to the specified position.

Wave_read.**tell**()
Return current file pointer position.

23.4.2 Objetos Wave_write

For seekable output streams, the wave header will automatically be updated to reflect the number of frames actually written. For unseekable streams, the *nframes* value must be accurate when the first frame data is written. An accurate *nframes* value can be achieved either by calling *setnframes()* or *setparams()* with the number of frames that will be written before *close()* is called and then using *writeframesraw()* to write the frame data, or by calling *writeframes()* with all of the frame data to be written. In the latter case *writeframes()* will calculate the number of frames in the data and set *nframes* accordingly before writing the frame data.

Wave_write objects, as returned by *open()*, have the following methods:

Alterado na versão 3.4: Added support for unseekable files.

Wave_write.**close**()
Make sure *nframes* is correct, and close the file if it was opened by *wave*. This method is called upon object collection. It will raise an exception if the output stream is not seekable and *nframes* does not match the number of frames actually written.

Wave_write.**setnchannels**(*n*)
Define o número de canais.

Wave_write.**setsampwidth**(*n*)
Set the sample width to *n* bytes.

Wave_write.**setframerate**(*n*)
Set the frame rate to *n*.

Alterado na versão 3.2: A non-integral input to this method is rounded to the nearest integer.

Wave_write.**setnframes**(*n*)
Set the number of frames to *n*. This will be changed later if the number of frames actually written is different (this update attempt will raise an error if the output stream is not seekable).

Wave_write.**setcomptype**(*type*, *name*)
Set the compression type and description. At the moment, only compression type NONE is supported, meaning no compression.

Wave_write.**setparams**(*tuple*)
The *tuple* should be (nchannels, sampwidth, framerate, nframes, comptype, compname), with values valid for the set*() methods. Sets all parameters.

Wave_write.**tell**()
Return current position in the file, with the same disclaimer for the *Wave_read.tell()* and *Wave_read.setpos()* methods.

Wave_write.**writeframesraw**(*data*)
Escreve quadros de áudio, sem corrigir *nframes*.

Alterado na versão 3.4: Todo objeto do tipo byte agora é aceito.

Wave_write.**writeframes**(*data*)
Write audio frames and make sure *nframes* is correct. It will raise an error if the output stream is not seekable and the total number of frames that have been written after *data* has been written does not match the previously set value for *nframes*.

Alterado na versão 3.4: Todo objeto do tipo byte agora é aceito.

Note that it is invalid to set any parameters after calling writeframes() or writeframesraw(), and any attempt to do so will raise *wave.Error*.

23.5 chunk — Read IFF chunked data

Código Fonte: Lib/chunk.py

This module provides an interface for reading files that use EA IFF 85 chunks.[1] This format is used in at least the Audio Interchange File Format (AIFF/AIFF-C) and the Real Media File Format (RMFF). The WAVE audio file format is closely related and can also be read using this module.

A chunk has the following structure:

Offset	Length	Conteúdo
0	4	Chunk ID
4	4	Size of chunk in big-endian byte order, not including the header
8	*n*	Data bytes, where *n* is the size given in the preceding field
8 + *n*	0 ou 1	Pad byte needed if *n* is odd and chunk alignment is used

The ID is a 4-byte string which identifies the type of chunk.

The size field (a 32-bit value, encoded using big-endian byte order) gives the size of the chunk data, not including the 8-byte header.

Usually an IFF-type file consists of one or more chunks. The proposed usage of the *Chunk* class defined here is to instantiate an instance at the start of each chunk and read from the instance until it reaches the end, after which a new instance can be instantiated. At the end of the file, creating a new instance will fail with an *EOFError* exception.

[1] "EA IFF 85" Standard for Interchange Format Files, Jerry Morrison, Electronic Arts, January 1985.

class chunk.**Chunk**(*file*, *align=True*, *bigendian=True*, *inclheader=False*)

Class which represents a chunk. The *file* argument is expected to be a file-like object. An instance of this class is specifically allowed. The only method that is needed is `read()`. If the methods *seek()* and *tell()* are present and don't raise an exception, they are also used. If these methods are present and raise an exception, they are expected to not have altered the object. If the optional argument *align* is true, chunks are assumed to be aligned on 2-byte boundaries. If *align* is false, no alignment is assumed. The default value is true. If the optional argument *bigendian* is false, the chunk size is assumed to be in little-endian order. This is needed for WAVE audio files. The default value is true. If the optional argument *inclheader* is true, the size given in the chunk header includes the size of the header. The default value is false.

A *Chunk* object supports the following methods:

getname()

Returns the name (ID) of the chunk. This is the first 4 bytes of the chunk.

getsize()

Returns the size of the chunk.

close()

Close and skip to the end of the chunk. This does not close the underlying file.

The remaining methods will raise *OSError* if called after the *close()* method has been called. Before Python 3.3, they used to raise *IOError*, now an alias of *OSError*.

isatty()

Returns `False`.

seek(*pos*, *whence=0*)

Set the chunk's current position. The *whence* argument is optional and defaults to `0` (absolute file positioning); other values are `1` (seek relative to the current position) and `2` (seek relative to the file's end). There is no return value. If the underlying file does not allow seek, only forward seeks are allowed.

tell()

Return the current position into the chunk.

read(*size=-1*)

Read at most *size* bytes from the chunk (less if the read hits the end of the chunk before obtaining *size* bytes). If the *size* argument is negative or omitted, read all data until the end of the chunk. An empty bytes object is returned when the end of the chunk is encountered immediately.

skip()

Skip to the end of the chunk. All further calls to *read()* for the chunk will return `b''`. If you are not interested in the contents of the chunk, this method should be called so that the file points to the start of the next chunk.

23.6 colorsys — Conversões entre sistemas de cores

Código Fonte: Lib/colorsys.py

O módulo *colorsys* define conversões bidirecionais de valores de cores entre cores expressas no espaço de cores RGB (Red Green Blue) usado em monitores de computador e três outros sistemas de coordenadas: YIQ, HLS (Hue Lightness Saturation) e HSV (Hue Saturation Value). As coordenadas em todos esses espaços de cores são valores de ponto flutuante. No espaço YIQ, a coordenada Y está entre 0 e 1, mas as coordenadas I e Q podem ser positivas ou negativas. Em todos os outros espaços, as coordenadas estão todas entre 0 e 1.

Ver também:

Mais informações sobre espaços de cores podem ser encontradas em http://poynton.ca/ColorFAQ.html e https://www.cambridgeincolour.com/tutorials/color-spaces.htm.

O módulo *colorsys* define as seguintes funções:

colorsys.**rgb_to_yiq**(*r*, *g*, *b*)
Converte a cor de coordenadas RGB para coordenadas YIQ.

colorsys.**yiq_to_rgb**(*y*, *i*, *q*)
Converte a cor de coordenadas YIQ para coordenadas RGB.

colorsys.**rgb_to_hls**(*r*, *g*, *b*)
Converte a cor de coordenadas RGB para coordenadas HLS.

colorsys.**hls_to_rgb**(*h*, *l*, *s*)
Converte a cor de coordenadas HLS para coordenadas RGB.

colorsys.**rgb_to_hsv**(*r*, *g*, *b*)
Converte a cor de coordenadas RGB para coordenadas HSV.

colorsys.**hsv_to_rgb**(*h*, *s*, *v*)
Converte a cor de coordenadas HSV para coordenadas RGB.

Exemplo:

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

23.7 imghdr — Determina o tipo de uma imagem

Código Fonte: Lib/imghdr.py

The *imghdr* module determines the type of image contained in a file or byte stream.

The *imghdr* module defines the following function:

imghdr.**what**(*filename*, *h=None*)
Tests the image data contained in the file named by *filename*, and returns a string describing the image type. If optional *h* is provided, the *filename* is ignored and *h* is assumed to contain the byte stream to test.

Alterado na versão 3.6: Aceita um *path-like object*.

The following image types are recognized, as listed below with the return value from *what()*:

Valor	Image format
`'rgb'`	SGI ImgLib Files
`'gif'`	GIF 87a and 89a Files
`'pbm'`	Portable Bitmap Files
`'pgm'`	Portable Graymap Files
`'ppm'`	Portable Pixmap Files
`'tiff'`	Arquivos TIFF
`'rast'`	Sun Raster Files
`'xbm'`	X Bitmap Files
`'jpeg'`	JPEG data in JFIF or Exif formats
`'bmp'`	BMP files
`'png'`	Portable Network Graphics
`'webp'`	WebP files
`'exr'`	OpenEXR Files

Novo na versão 3.5: The *exr* and *webp* formats were added.

You can extend the list of file types *imghdr* can recognize by appending to this variable:

imghdr.**tests**
: A list of functions performing the individual tests. Each function takes two arguments: the byte-stream and an open file-like object. When *what()* is called with a byte-stream, the file-like object will be `None`.

  A função de teste deve retornar uma string descrevendo o tipo de imagem, se o teste for bem-sucedido, ou `None`, se falhar.

Exemplo:

```
>>> import imghdr
>>> imghdr.what('bass.gif')
'gif'
```

23.8 sndhdr — Determina o tipo de arquivos de som

Código Fonte: Lib/sndhdr.py

O *sndhdr* fornece funções de utilitário que tentam determinar o tipo de dados de som que estão em um arquivo. Quando estas funções são capazes de determinar que tipo de dados de som são armazenados em um arquivo, eles retornam um *namedtuple()*, contendo cinco atributos: (`filetype`, `framerate`, `nchannels`, `nframes`, `sampwidth`). O valor para *type* indica o tipo de dados e será uma das strings `'aifc'`, `'aiff'`, `'au'`, `'hcom'`, `'sndr'`, `'sndt'`, `'voc'`, `'wav'`, `'8svx'`, `'sb'`, `'ub'`, or `'ul'`. O *sample_rate* será o valor real ou `0` se desconhecido ou difícil de decodificar. Similarmente, *channels* será o número de canais ou `0` se não puder ser determinado ou se o valor for difícil de decodificar. O valor para *frames* será o número de quadros ou `-1`. O último item na tupla, *bits_per_sample*, será o tamanho da amostra em bits ou `'A'` para A-LAW ou `'U'` para u-LAW.

sndhdr.**what**(*filename*)
: Determina o tipo de dados de som armazenados no arquivo *filename* usando *whathdr()*. Se tiver sucesso, retorna uma namedtuple conforme descrito acima, caso contrário, `None` será retornado.

  Alterado na versão 3.5: Resultado alterado de uma tupla para uma namedtuple.

sndhdr.**whathdr**(*filename*)
Determina o tipo de dados de som armazenados em um arquivo com base no cabeçalho do arquivo. O nome do arquivo é dado por *filename*. Esta função retorna um namedtuple como descrito acima ao obter sucesso, ou None.

Alterado na versão 3.5: Resultado alterado de uma tupla para uma namedtuple.

23.9 ossaudiodev — Access to OSS-compatible audio devices

This module allows you to access the OSS (Open Sound System) audio interface. OSS is available for a wide range of open-source and commercial Unices, and is the standard audio interface for Linux and recent versions of FreeBSD.

Alterado na versão 3.3: As operações neste módulo agora levantam *OSError* onde *IOError* foi levantado.

Ver também:

Open Sound System Programmer's Guide the official documentation for the OSS C API

The module defines a large number of constants supplied by the OSS device driver; see <sys/soundcard.h> on either Linux or FreeBSD for a listing.

ossaudiodev defines the following variables and functions:

exception ossaudiodev.**OSSAudioError**
This exception is raised on certain errors. The argument is a string describing what went wrong.

(If *ossaudiodev* receives an error from a system call such as open(), write(), or ioctl(), it raises *OSError*. Errors detected directly by *ossaudiodev* result in *OSSAudioError*.)

(For backwards compatibility, the exception class is also available as ossaudiodev.error.)

ossaudiodev.**open**(*mode*)
ossaudiodev.**open**(*device*, *mode*)
Open an audio device and return an OSS audio device object. This object supports many file-like methods, such as read(), write(), and fileno() (although there are subtle differences between conventional Unix read/write semantics and those of OSS audio devices). It also supports a number of audio-specific methods; see below for the complete list of methods.

device is the audio device filename to use. If it is not specified, this module first looks in the environment variable AUDIODEV for a device to use. If not found, it falls back to /dev/dsp.

mode is one of 'r' for read-only (record) access, 'w' for write-only (playback) access and 'rw' for both. Since many sound cards only allow one process to have the recorder or player open at a time, it is a good idea to open the device only for the activity needed. Further, some sound cards are half-duplex: they can be opened for reading or writing, but not both at once.

Note the unusual calling syntax: the *first* argument is optional, and the second is required. This is a historical artifact for compatibility with the older linuxaudiodev module which *ossaudiodev* supersedes.

ossaudiodev.**openmixer**([*device*])
Open a mixer device and return an OSS mixer device object. *device* is the mixer device filename to use. If it is not specified, this module first looks in the environment variable MIXERDEV for a device to use. If not found, it falls back to /dev/mixer.

23.9.1 Audio Device Objects

Before you can write to or read from an audio device, you must call three methods in the correct order:

1. `setfmt()` to set the output format
2. `channels()` to set the number of channels
3. `speed()` to set the sample rate

Alternately, you can use the `setparameters()` method to set all three audio parameters at once. This is more convenient, but may not be as flexible in all cases.

The audio device objects returned by *open()* define the following methods and (read-only) attributes:

`oss_audio_device.`**`close`**`()`
Explicitly close the audio device. When you are done writing to or reading from an audio device, you should explicitly close it. A closed device cannot be used again.

`oss_audio_device.`**`fileno`**`()`
Return the file descriptor associated with the device.

`oss_audio_device.`**`read`**`(`*size*`)`
Read *size* bytes from the audio input and return them as a Python string. Unlike most Unix device drivers, OSS audio devices in blocking mode (the default) will block *read()* until the entire requested amount of data is available.

`oss_audio_device.`**`write`**`(`*data*`)`
Write a *bytes-like object* *data* to the audio device and return the number of bytes written. If the audio device is in blocking mode (the default), the entire data is always written (again, this is different from usual Unix device semantics). If the device is in non-blocking mode, some data may not be written—see *writeall()*.

Alterado na versão 3.5: Writable *bytes-like object* is now accepted.

`oss_audio_device.`**`writeall`**`(`*data*`)`
Write a *bytes-like object* *data* to the audio device: waits until the audio device is able to accept data, writes as much data as it will accept, and repeats until *data* has been completely written. If the device is in blocking mode (the default), this has the same effect as *write()*; *writeall()* is only useful in non-blocking mode. Has no return value, since the amount of data written is always equal to the amount of data supplied.

Alterado na versão 3.5: Writable *bytes-like object* is now accepted.

Alterado na versão 3.2: Audio device objects also support the context management protocol, i.e. they can be used in a `with` statement.

Os seguintes métodos são mapeados para exatamente uma chamada de sistema `ioctl()`. A correspondência é óbvia: por exemplo, `setfmt()` corresponde a `SNDCTL_DSP_SETFMT` ioctl e `sync()` a `SNDCTL_DSP_SYNC` (isto pode ser útil ao consultar a documentação do OSS). Se o `ioctl()` subjacente falhar, todos eles levantam *OSError*.

`oss_audio_device.`**`nonblock`**`()`
Put the device into non-blocking mode. Once in non-blocking mode, there is no way to return it to blocking mode.

`oss_audio_device.`**`getfmts`**`()`
Return a bitmask of the audio output formats supported by the soundcard. Some of the formats supported by OSS are:

Formatação	Description (descrição)
AFMT_MU_LAW	a logarithmic encoding (used by Sun .au files and /dev/audio)
AFMT_A_LAW	a logarithmic encoding
AFMT_IMA_ADPCM	a 4:1 compressed format defined by the Interactive Multimedia Association
AFMT_U8	Unsigned, 8-bit audio
AFMT_S16_LE	Signed, 16-bit audio, little-endian byte order (as used by Intel processors)
AFMT_S16_BE	Signed, 16-bit audio, big-endian byte order (as used by 68k, PowerPC, Sparc)
AFMT_S8	Signed, 8 bit audio
AFMT_U16_LE	Unsigned, 16-bit little-endian audio
AFMT_U16_BE	Unsigned, 16-bit big-endian audio

Consult the OSS documentation for a full list of audio formats, and note that most devices support only a subset of these formats. Some older devices only support AFMT_U8; the most common format used today is AFMT_S16_LE.

oss_audio_device.**setfmt**(*format*)
Try to set the current audio format to *format*—see *getfmts()* for a list. Returns the audio format that the device was set to, which may not be the requested format. May also be used to return the current audio format—do this by passing an "audio format" of AFMT_QUERY.

oss_audio_device.**channels**(*nchannels*)
Set the number of output channels to *nchannels*. A value of 1 indicates monophonic sound, 2 stereophonic. Some devices may have more than 2 channels, and some high-end devices may not support mono. Returns the number of channels the device was set to.

oss_audio_device.**speed**(*samplerate*)
Try to set the audio sampling rate to *samplerate* samples per second. Returns the rate actually set. Most sound devices don't support arbitrary sampling rates. Common rates are:

Rate	Description (descrição)
8000	default rate for /dev/audio
11025	speech recording
22050	
44100	CD quality audio (at 16 bits/sample and 2 channels)
96000	DVD quality audio (at 24 bits/sample)

oss_audio_device.**sync**()
Wait until the sound device has played every byte in its buffer. (This happens implicitly when the device is closed.) The OSS documentation recommends closing and re-opening the device rather than using *sync()*.

oss_audio_device.**reset**()
Immediately stop playing or recording and return the device to a state where it can accept commands. The OSS documentation recommends closing and re-opening the device after calling *reset()*.

oss_audio_device.**post**()
Tell the driver that there is likely to be a pause in the output, making it possible for the device to handle the pause more intelligently. You might use this after playing a spot sound effect, before waiting for user input, or before doing disk I/O.

The following convenience methods combine several ioctls, or one ioctl and some simple calculations.

oss_audio_device.**setparameters**(*format*, *nchannels*, *samplerate*[, *strict=False*])
Set the key audio sampling parameters—sample format, number of channels, and sampling rate—in one method call. *format*, *nchannels*, and *samplerate* should be as specified in the *setfmt()*, *channels()*, and *speed()* methods. If *strict* is true, *setparameters()* checks to see if each parameter was actually set to the requested value, and raises *OSSAudioError* if not. Returns a tuple (*format*, *nchannels*, *samplerate*) indicating the

parameter values that were actually set by the device driver (i.e., the same as the return values of *setfmt()*, *channels()*, and *speed()*).

Por exemplo:

```
(fmt, channels, rate) = dsp.setparameters(fmt, channels, rate)
```

é equivalente a:

```
fmt = dsp.setfmt(fmt)
channels = dsp.channels(channels)
rate = dsp.rate(rate)
```

oss_audio_device.**bufsize**()
Returns the size of the hardware buffer, in samples.

oss_audio_device.**obufcount**()
Returns the number of samples that are in the hardware buffer yet to be played.

oss_audio_device.**obuffree**()
Returns the number of samples that could be queued into the hardware buffer to be played without blocking.

Audio device objects also support several read-only attributes:

oss_audio_device.**closed**
Boolean indicating whether the device has been closed.

oss_audio_device.**name**
String containing the name of the device file.

oss_audio_device.**mode**
The I/O mode for the file, either `"r"`, `"rw"`, or `"w"`.

23.9.2 Mixer Device Objects

The mixer object provides two file-like methods:

oss_mixer_device.**close**()
This method closes the open mixer device file. Any further attempts to use the mixer after this file is closed will raise an *OSError*.

oss_mixer_device.**fileno**()
Returns the file handle number of the open mixer device file.

Alterado na versão 3.2: Mixer objects also support the context management protocol.

Os métodos restantes são específicos para edição de audio:

oss_mixer_device.**controls**()
This method returns a bitmask specifying the available mixer controls ("Control" being a specific mixable "channel", such as `SOUND_MIXER_PCM` or `SOUND_MIXER_SYNTH`). This bitmask indicates a subset of all available mixer controls—the `SOUND_MIXER_*` constants defined at module level. To determine if, for example, the current mixer object supports a PCM mixer, use the following Python code:

```
mixer=ossaudiodev.openmixer()
if mixer.controls() & (1 << ossaudiodev.SOUND_MIXER_PCM):
    # PCM is supported
    ... code ...
```

For most purposes, the SOUND_MIXER_VOLUME (master volume) and SOUND_MIXER_PCM controls should suffice—but code that uses the mixer should be flexible when it comes to choosing mixer controls. On the Gravis Ultrasound, for example, SOUND_MIXER_VOLUME does not exist.

oss_mixer_device.**stereocontrols**()
Returns a bitmask indicating stereo mixer controls. If a bit is set, the corresponding control is stereo; if it is unset, the control is either monophonic or not supported by the mixer (use in combination with *controls()* to determine which).

See the code example for the *controls()* function for an example of getting data from a bitmask.

oss_mixer_device.**reccontrols**()
Returns a bitmask specifying the mixer controls that may be used to record. See the code example for *controls()* for an example of reading from a bitmask.

oss_mixer_device.**get**(*control*)
Returns the volume of a given mixer control. The returned volume is a 2-tuple (left_volume, right_volume). Volumes are specified as numbers from 0 (silent) to 100 (full volume). If the control is monophonic, a 2-tuple is still returned, but both volumes are the same.

Raises *OSSAudioError* if an invalid control is specified, or *OSError* if an unsupported control is specified.

oss_mixer_device.**set**(*control*, *(left*, *right)*)
Sets the volume for a given mixer control to (left,right). left and right must be ints and between 0 (silent) and 100 (full volume). On success, the new volume is returned as a 2-tuple. Note that this may not be exactly the same as the volume specified, because of the limited resolution of some soundcard's mixers.

Raises *OSSAudioError* if an invalid mixer control was specified, or if the specified volumes were out-of-range.

oss_mixer_device.**get_recsrc**()
This method returns a bitmask indicating which control(s) are currently being used as a recording source.

oss_mixer_device.**set_recsrc**(*bitmask*)
Call this function to specify a recording source. Returns a bitmask indicating the new recording source (or sources) if successful; raises *OSError* if an invalid source was specified. To set the current recording source to the microphone input:

```
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```

CAPÍTULO 24

Internacionalização

Os módulos descritos neste capítulo ajudam você a criar um software que é independente de idioma e localidade, fornecendo mecanismos para selecionar um idioma a ser usado em mensagens de programa ou adaptando a saída para corresponder às convenções locais.

A lista de módulos descritos neste capítulo é:

24.1 `gettext` — Serviços de internacionalização multilíngues

Código Fonte: Lib/gettext.py

O módulo *gettext* fornece serviços de internacionalização (I18N) e localização (L10N) para seus módulos e aplicativos Python. Ele suporta a API do catálogo de mensagens GNU **gettext** e uma API baseada em classes de nível mais alto que podem ser mais apropriadas para arquivos Python. A interface descrita abaixo permite gravar o módulo e as mensagens do aplicativo em um idioma natural e fornecer um catálogo de mensagens traduzidas para execução em diferentes idiomas naturais.

Algumas dicas sobre localização de seus módulos e aplicativos Python também são fornecidas.

24.1.1 API do GNU `gettext`

O módulo *gettext* define a API a seguir, que é muito semelhante à API do GNU **gettext**. Se você usar esta API, você afetará a tradução de todo o seu aplicativo globalmente. Geralmente, é isso que você deseja se o seu aplicativo for monolíngue, com a escolha do idioma dependente da localidade do seu usuário. Se você estiver localizando um módulo Python, ou se seu aplicativo precisar alternar idiomas rapidamente, provavelmente desejará usar a API baseada em classe.

gettext.**bindtextdomain**(*domain*, *localedir=None*)

Bind the *domain* to the locale directory *localedir*. More concretely, *gettext* will look for binary `.mo` files for the given domain using the path (on Unix): *localedir*/*language*/LC_MESSAGES/*domain*.mo, where

languages is searched for in the environment variables `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG` respectively.

Se *localedir* for omitido ou `None`, a ligação atual para *domain* será retornada.[1]

`gettext.`**`bind_textdomain_codeset`**(*domain*, *codeset=None*)
Liga o *domain* ao *codeset*, alterando a codificação das strings de bytes retornadas pelas funções *lgettext()*, *ldgettext()*, *lngettext()* e *ldngettext()*. Se *codeset* for omitido, a ligação atual será retornada.

`gettext.`**`textdomain`**(*domain=None*)
Altera ou consulta o domínio global atual. Se *domain* for `None`, o domínio global atual será retornado; caso contrário, o domínio global será definido como *domain*, que será retornado.

`gettext.`**`gettext`**(*message*)
Retorna a tradução localizada de *message*, com base no diretório global atual de domínio, idioma e localidade. Essa função geralmente é apelidada como `_()` no espaço de nomes local (veja exemplos abaixo).

`gettext.`**`dgettext`**(*domain*, *message*)
Semelhante a *gettext()*, mas procura a mensagem no *domain* especificado.

`gettext.`**`ngettext`**(*singular*, *plural*, *n*)
Semelhante a *gettext()*, mas considera formas plurais. Se uma tradução for encontrada, aplica a fórmula do plural a *n* e retorne a mensagem resultante (alguns idiomas têm mais de duas formas no plural). Se nenhuma tradução for encontrada, retorna *singular* se *n* for 1; retorna *plural* caso contrário.

A fórmula de Plural é retirada do cabeçalho do catálogo. É uma expressão C ou Python que possui uma variável livre *n*; a expressão é avaliada para o índice do plural no catálogo. Veja a documentação do gettext GNU para obter a sintaxe precisa a ser usada em arquivos `.po` e as fórmulas para um variedade de idiomas.

`gettext.`**`dngettext`**(*domain*, *singular*, *plural*, *n*)
Semelhante a *ngettext()*, mas procura mensagens no *domain* especificado.

`gettext.`**`lgettext`**(*message*)

`gettext.`**`ldgettext`**(*domain*, *message*)

`gettext.`**`lngettext`**(*singular*, *plural*, *n*)

`gettext.`**`ldngettext`**(*domain*, *singular*, *plural*, *n*)
Equivalente às funções correspondentes sem o prefixo `l` (*gettext()*, *dgettext()*, *ngettext()* e *dngettext()*), mas a tradução é retornada como uma string de bytes codificada na codificação preferida do sistema, se nenhuma outra codificação foi definida explicitamente com *bind_textdomain_codeset()*.

> **Aviso:** These functions should be avoided in Python 3, because they return encoded bytes. It's much better to use alternatives which return Unicode strings instead, since most Python applications will want to manipulate human readable text as strings instead of bytes. Further, it's possible that you may get unexpected Unicode-related exceptions if there are encoding problems with the translated strings. It is possible that the `l*()` functions will be deprecated in future Python versions due to their inherent problems and limitations.

Note que GNU **`gettext`** também define um método `dcgettext()`, mas isso não foi considerado útil e, portanto, atualmente não está implementado.

Aqui está um exemplo de uso típico para esta API:

[1] O diretório de localidade padrão depende do sistema; por exemplo, no RedHat Linux é `/usr/share/locale`, mas no Solaris é `/usr/lib/locale`. O módulo *gettext* não tenta dar suporte a esses padrões dependentes do sistema; em vez disso, seu padrão é *sys.base_prefix*`/share/locale` (consulte *sys.base_prefix*). Por esse motivo, é sempre melhor chamar *bindtextdomain()* com um caminho absoluto explícito no início da sua aplicação.

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print(_('This is a translatable string.'))
```

24.1.2 API baseada em classe

A API baseada em classe do módulo *gettext* oferece mais flexibilidade e maior conveniência do que a API do GNU **gettext**. É a maneira recomendada de localizar seus aplicativos e módulos Python. `gettext` define uma classe *GNUTranslations* que implementa a análise de arquivos no formato GNU `.mo` e possui métodos para retornar strings. Instâncias dessa classe também podem se instalar no espaço de nomes interno como a função `_()`.

`gettext.`**`find`**(*domain*, *localedir=None*, *languages=None*, *all=False*)

Esta função implementa o algoritmo de busca de arquivos `.mo` padrão. É necessário um *domain*, idêntico ao que *textdomain()* leva. *localedir* opcional é como em *bindtextdomain()*. *languages* opcional é uma lista de strings, em que cada string é um código de idioma.

Se *localedir* não for fornecido, o diretório local do sistema padrão será usado.[2] Se *languages* não for fornecido, as seguintes variáveis de ambiente serão pesquisadas: `LANGUAGE`, `LC_ALL`, `LC_MESSAGES` e `LANG`. O primeiro retornando um valor não vazio é usado para a variável *languages*. As variáveis de ambiente devem conter uma lista de idiomas separada por dois pontos, que será dividida nos dois pontos para produzir a lista esperada de cadeias de código de idioma.

find() expande e normaliza os idiomas e itera através deles, procurando por um arquivo existente construído com esses componentes:

localedir`/`*language*`/LC_MESSAGES/`*domain*`.mo`

O primeiro nome de arquivo existente é retornado por *find()*. Se nenhum desses arquivos for encontrado, será retornado `None`. Se *all* for fornecido, ele retornará uma lista de todos os nomes de arquivos, na ordem em que aparecem na lista de idiomas ou nas variáveis de ambiente.

`gettext.`**`translation`**(*domain*, *localedir=None*, *languages=None*, *class_=None*, *fallback=False*, *codeset=None*)

Retorna uma instância de `*Translations` com base nos *domain*, *localedir* e *languages*, que são passados primeiro para *find()* para obter uma lista dos caminhos de arquivos `.mo` associados. Instâncias com nomes de arquivo idênticos `.mo` são armazenados em cache. A classe atual instanciada é *class_* se fornecida, caso contrário *GNUTranslations*. O construtor da classe deve usar um único argumento objeto arquivo. Se fornecido, *codeset* alterará o conjunto de caracteres usado para codificar as strings traduzidas nos métodos *lgettext()* e *lngettext()*.

Se vários arquivos forem encontrados, os arquivos posteriores serão usados como fallbacks para os anteriores. Para permitir a configuração do fallback, *copy.copy()* é usado para clonar cada objeto de conversão do cache; os dados reais da instância ainda são compartilhados com o cache.

Se nenhum arquivo `.mo` for encontrado, essa função levanta *OSError* se *fallback* for falso (que é o padrão) e retorna uma instância *NullTranslations* se *fallback* for verdadeiro.

Alterado na versão 3.3: *IOError* costumava ser levantado em vez do *OSError*.

`gettext.`**`install`**(*domain*, *localedir=None*, *codeset=None*, *names=None*)

Isso instala a função `_()` no espaço de nomes interno do Python, com base em *domain*, *localedir* e *codeset* que são passados para a função *translation()*.

Para o parâmetro *names*, por favor, veja a descrição método *install()* do objeto de tradução.

[2] Consulte a nota de rodapé para a *bindtextdomain()* acima.

Como visto abaixo, você normalmente marca as strings candidatas à tradução em seu aplicativo, envolvendo-as em uma chamada para a função `_()`, assim:

```
print(_('This string will be translated.'))
```

Por conveniência, você deseja que a função `_()` seja instalada no espaço de nomes interno do Python, para que seja facilmente acessível em todos os módulos do seu aplicativo.

A classe `NullTranslations`

As classes de tradução são o que realmente implementa a tradução de strings de mensagens do arquivo-fonte original para strings de mensagens traduzidas. A classe base usada por todas as classes de tradução é *`NullTranslations`*; isso fornece a interface básica que você pode usar para escrever suas próprias classes de tradução especializadas. Aqui estão os métodos de `NullTranslations`:

class gettext.**NullTranslations**(*fp=None*)

Recebe um *objeto arquivo* opcional *fp*, que é ignorado pela classe base. Inicializa as variáveis de instância "protegidas" *_info* e *_charset*, que são definidas por classes derivadas, bem como *_fallback*, que é definido através de *`add_fallback()`*. Ele então chama `self._parse(fp)` se *fp* não for `None`.

_parse(*fp*)

No-op na classe base, esse método pega o objeto de arquivo *fp* e lê os dados do arquivo, inicializando seu catálogo de mensagens. Se você tiver um formato de arquivo de catálogo de mensagens não suportado, substitua esse método para analisar seu formato.

add_fallback(*fallback*)

Adiciona *fallback* como o objeto reserva para o objeto de tradução atual. Um objeto de tradução deve consultar o fallback se não puder fornecer uma tradução para uma determinada mensagem.

gettext(*message*)

Se um fallback tiver sido definido, encaminhe `Gettext()` para o fallback. Caso contrário, retorne *message*. Sobrecarregado em classes derivadas.

ngettext(*singular*, *plural*, *n*)

Se um fallback tiver sido definido, encaminha `ngettext()` para o fallback. Caso contrário, retorne *singular* se *n* for 1; do contrário, retorna *plural*. Sobrecarregado em classes derivadas.

lgettext(*message*)

lngettext(*singular*, *plural*, *n*)

Equivale a *`gettext()`* e *`ngettext()`*, mas a tradução é retornada como uma string de bytes codificada na codificação preferida do sistema, se nenhuma codificação foi explicitamente definida com *`set_output_charset()`*. Sobrecarregado em classes derivadas.

> **Aviso:** Esses métodos devem ser evitados no Python 3. Veja o aviso para a função *`lgettext()`*.

info()

Retorna a variável `_info` "protegida", um dicionário que contém os metadados encontrados no arquivo de catálogo de mensagens.

charset()

Retorna a codificação do arquivo de catálogo de mensagens.

output_charset()

Retorna a codificação usada para retornar as mensagens traduzidas em *`lgettext()`* e *`lngettext()`*.

set_output_charset(*charset*)
Altera a codificação usada para retornar mensagens traduzidas.

install(*names=None*)
Este método instala *gettext()* no espaço de nomes embutido, vinculando-o a _.

If the *names* parameter is given, it must be a sequence containing the names of functions you want to install in the builtins namespace in addition to _(). Supported names are 'gettext', 'ngettext', 'lgettext' and 'lngettext'.

Observe que esta é apenas uma maneira, embora a maneira mais conveniente, de disponibilizar a função _() para o seu aplicativo. Como afeta o aplicativo inteiro globalmente, e especificamente o espaço de nomes embutido, os módulos localizados nunca devem instalar _(). Em vez disso, eles devem usar este código para disponibilizar _() para seu módulo:

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

Isso coloca _() apenas no espaço de nomes global do módulo e, portanto, afeta apenas as chamadas dentro deste módulo.

A classe GNUTranslations

O módulo *gettext* fornece uma classe adicional derivada de *NullTranslations*: *GNUTranslations*. Esta classe sobrecarrega _parse() para permitir a leitura de arquivos .mo do formato GNU **gettext** nos formatos big-endian e little-endian.

GNUTranslations analisa metadados opcionais do catálogo de tradução. É uma convenção com o GNU **gettext** incluir metadados como tradução para a string vazia. Esses metadados estão nos pares chave: valor no estilo **RFC 822** e devem conter a chave Project-Id-Version. Se a chave Content-Type for encontrada, a propriedade charset será usada para inicializar a variável de instância _charset "protegida", com o padrão None se não for encontrada. Se a codificação de "charset" for especificada, todos os IDs e strings de mensagens lidos no catálogo serão convertidos em Unicode usando essa codificação, caso contrário, o ASCII será presumido.

Como os IDs de mensagens também são lidos como strings Unicode, todos os métodos *gettext() presumem os IDs de mensagens como sendo strings Unicode, não como strings de bytes.

Todo o conjunto de pares chave/valor é colocado em um dicionário e definido como a variável de instância _info "protegida".

Se o número mágico do arquivo .mo for inválido, o número principal da versão é inesperado ou se ocorrerem outros problemas durante a leitura do arquivo, instanciando uma classe *GNUTranslations* pode levantar *OSError*.

class gettext.**GNUTranslations**
Os seguintes métodos são substituídos a partir da implementação da classe base:

gettext(*message*)
Procura o ID da *message* no catálogo e retorna a sequência de mensagens correspondente, como uma string Unicode. Se não houver entrada no catálogo para o ID da *message* e um fallback tiver sido definido, a pesquisa será encaminhada para o método do fallback *gettext()*. Caso contrário, o ID da *message* é retornado.

ngettext(*singular*, *plural*, *n*)
faz uma pesquisa de plural-forms de um ID de mensagem. *singular* é usado como o ID da mensagem para fins de pesquisa no catálogo, enquanto *n* é usado para determinar qual forma plural usar. A string de mensagens retornada é uma string Unicode.

Se o ID da mensagem não for encontrado no catálogo e um fallback for especificado, a solicitação será encaminhada para o método do fallback *ngettext()*. Caso contrário, quando *n* for 1, *singular* será retornado e *plural* será retornado em todos os outros casos.

Aqui está um exemplo:

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ngettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

lgettext(*message*)

lngettext(*singular*, *plural*, *n*)

Equivale a *gettext()* e *ngettext()*, mas a tradução é retornada como uma string de bytes codificada na codificação preferida do sistema, se nenhuma codificação foi explicitamente definida com *set_output_charset()*.

Aviso: Esses métodos devem ser evitados no Python 3. Veja o aviso para a função *lgettext()*.

Suporte a catálogo de mensagens do Solaris

O sistema operacional Solaris define seu próprio formato de arquivo binário `.mo`, mas como nenhuma documentação pode ser encontrada nesse formato, ela não é suportada no momento.

O construtor Catalog

O GNOME usa uma versão do módulo *gettext* de James Henstridge, mas esta versão tem uma API um pouco diferente. Seu uso documentado foi:

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print(_('hello world'))
```

Para compatibilidade com este módulo mais antigo, a função `Catalog()` é um apelido para a função *translation()* descrita acima.

Uma diferença entre este módulo e o de Henstridge: seus objetos de catálogo suportavam o acesso por meio de uma API de mapeamento, mas isso parece não ser utilizado e, portanto, não é atualmente suportado.

24.1.3 Internacionalizando seus programas e módulos

Internationalization (I18N), ou internacionalização (I17O) em português, refere-se à operação pela qual um programa é informado sobre vários idiomas. Localization (L10N), ou localização em português, refere-se à adaptação do seu programa, uma vez internacionalizado, aos hábitos culturais e de idioma local. Para fornecer mensagens multilíngues para seus programas Python, você precisa executar as seguintes etapas:

1. preparar seu programa ou módulo marcando especialmente strings traduzíveis
2. executar um conjunto de ferramentas nos arquivos marcados para gerar catálogos de mensagens não tratadas

3. criar traduções específicas do idioma dos catálogos de mensagens
4. usar o módulo *gettext* para que as strings das mensagens sejam traduzidas corretamente

Para preparar seu código para I18N, você precisa examinar todas as strings em seus arquivos. Qualquer string que precise ser traduzida deve ser marcada envolvendo-a em `_('...')` — isto é, uma chamada para a função `_()`. Por exemplo:

```
filename = 'mylog.txt'
message = _('writing a log message')
with open(filename, 'w') as fp:
    fp.write(message)
```

Neste exemplo, a string `'writing a log message'` está marcada como um candidato para tradução, enquanto as strings `'mylog.txt'` e `'w'` não estão.

Existem algumas ferramentas para extrair as strings destinadas à tradução. O GNU **gettext** original tem suporte apenas ao código-fonte C ou C++, mas sua versão estendida **xgettext** varre o código escrito em várias linguagens, incluindo Python, para encontrar strings marcadas como traduzíveis. Babel é uma biblioteca de internacionalização do Python que inclui um script `pybabel` para extrair e compilar catálogos de mensagens. O programa de François Pinard chamado **xpot** faz um trabalho semelhante e está disponível como parte de seu pacote po-utils.

(O Python também inclui versões em Python puro desses programas, chamadas **pygettext.py** e **msgfmt.py**; algumas distribuições do Python as instalam para você. O **pygettext.py** é semelhante ao **xgettext**, mas apenas entende o código-fonte do Python e não consegue lidar com outras linguagens de programação como C ou C++. O **pygettext.py** possui suporte a uma interface de linha de comando semelhante à do **xgettext**; para detalhes sobre seu uso, execute `pygettext.py --help`. O **msgfmt.py** é compatível com binários com GNU **msgfmt**. Com esses dois programas, você pode não precisar do pacote GNU :program‘gettext‘ para internacionalizar seus aplicativos Python.)

xgettext, **pygettext** e ferramentas similares geram `.po` que são catálogos de mensagens. Eles são arquivos legíveis por humanos estruturados que contêm todas as strings marcadas no código-fonte, junto com um espaço reservado para as versões traduzidas dessas strings.

Cópias destes arquivos `.po` são entregues aos tradutores humanos individuais que escrevem traduções para todos os idiomas naturais suportados. Eles enviam de volta as versões completas específicas do idioma como um arquivo `<nome-do-idioma>.po` que é compilado em um arquivo de catálogo binário legível por máquina `.mo` usando o programa **msgfmt**. Os arquivos `.mo` são usados pelo módulo *gettext* para o processamento de tradução real em tempo de execução.

Como você usa o módulo *gettext* no seu código depende se você está internacionalizando um único módulo ou sua aplicação inteira. As próximas duas seções discutirão cada caso.

Localizando seu módulo

Se você estiver localizando seu módulo, tome cuidado para não fazer alterações globais, por exemplo para o espaço de nomes embutidos. Você não deve usar a API GNU **gettext**, mas a API baseada em classe.

Digamos que seu módulo seja chamado “spam” e as várias traduções do idioma natural do arquivo `.mo` residam em `/usr/share/locale` no formato GNU **gettext**. Aqui está o que você colocaria sobre o seu módulo:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.gettext
```

Localizando sua aplicação

Se você estiver localizando sua aplicação, poderá instalar a função `_()` globalmente no espaço de nomes embutidos, geralmente no arquivo principal do driver do sua aplicação. Isso permitirá que todos os arquivos específicos de sua aplicação usem `_('...')` sem precisar instalá-la explicitamente em cada arquivo.

No caso simples, você precisa adicionar apenas o seguinte código ao arquivo do driver principal da sua aplicação:

```
import gettext
gettext.install('myapplication')
```

Se você precisar definir o diretório da localidade, poderá passá-lo para a função *install()*:

```
import gettext
gettext.install('myapplication', '/usr/share/locale')
```

Alterando os idiomas durante o uso

Se o seu programa precisar oferecer suporte a vários idiomas ao mesmo tempo, convém criar várias instâncias de tradução e alternar entre elas explicitamente, assim:

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

Traduções adiadas

Na maioria das situações de codificação, as strings são traduzidas onde são codificadas. Ocasionalmente, no entanto, é necessário marcar strings para tradução, mas adiar a tradução real até mais tarde. Um exemplo clássico é:

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]
# ...
for a in animals:
    print(a)
```

Aqui, você deseja marcar as strings na lista `animals` como traduzíveis, mas na verdade não deseja traduzi-las até que sejam impressas.

Aqui está uma maneira de lidar com esta situação:

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print(_(a))
```

Isso funciona porque a definição fictícia de `_()` simplesmente retorna a string inalterada. E essa definição fictícia va substituir temporariamente qualquer definição de `_()` no espaço de nomes embutido (até o comando `del`). Tome cuidado, se você tiver uma definição anterior de `_()` no espaço de nomes local.

Observe que o segundo uso de `_()` não identificará "a" como traduzível para o programa **gettext**, porque o parâmetro não é uma string literal.

Outra maneira de lidar com isso é com o seguinte exemplo:

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print(_(a))
```

Nesse caso, você está marcando strings traduzíveis com a função `N_()`, que não entra em conflito com nenhuma definição de `_()`. No entanto, você precisará ensinar seu programa de extração de mensagens a procurar strings traduzíveis marcadas com `N_()`. **xgettext**, **pygettext**, `pybabel extract` e **xpot** possuem suporte a isso através do uso da opção de linha de comando `-k`. A escolha de `N_()` aqui é totalmente arbitrária; poderia facilmente ter sido `MarkThisStringForTranslation()`.

24.1.4 Reconhecimentos

As seguintes pessoas contribuíram com código, feedback, sugestões de design, implementações anteriores e experiência valiosa para a criação deste módulo:

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw

- Gustavo Niemeyer

24.2 `locale` — Serviços de internacionalização

Código Fonte: Lib/locale.py

O módulo *`locale`* abre o acesso ao banco de dados de localidade POSIX e funcionalidade. O mecanismo de localidade POSIX permite que os programadores lidem com certas questões culturais em uma aplicação, sem exigir que o programador conheça todas as especificidades de cada país onde o software é executado.

O módulo *`locale`* é implementado em cima do módulo `_locale`, que por sua vez usa uma implementação a localidade ANSI C se disponível.

O módulo *`locale`* define a seguinte exceção e funções:

exception locale.**Error**
Exceção levantada quando a localidade passada para *`setlocale()`* não é reconhecida.

locale.**setlocale**(*category*, *locale=None*)
Se *locale* for fornecido e não `None`, *`setlocale()`* modifica a configuração de locale para a *category*. As categorias disponíveis estão listadas na descrição dos dados abaixo. *locale* pode ser uma string ou um iterável de duas strings (código de idioma e codificação). Se for um iterável, ele é convertido em um nome de local usando o mecanismo de alias de local. Uma string vazia especifica as configurações padrão do usuário. Se a modificação da localidade falhar, a exceção *`Error`* é levantada. Se for bem-sucedido, a nova configuração de localidade será retornada.

Se *locale* for omitido ou `None`, a configuração atual for *category* é retornado.

`setlocale()` não é seguro para thread na maioria dos sistemas. As aplicações normalmente começam com uma chamada de

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

Isso define a localidade de todas as categorias para a configuração padrão do usuário (normalmente especificada na variável de ambiente `LANG`). Se a localidade não for alterada depois disso, o uso de multithreading não deve causar problemas.

locale.**localeconv**()
Retorna o banco de dados das convenções locais como um dicionário. Este dicionário possui as seguintes strings como chaves:

Categoria	Chave	Significado
LC_NUMERIC	'decimal_point'	Caractere de ponto decimal.
	'grouping'	Sequência de números especificando quais posições relativas os 'thousands_sep' são esperados. Se a sequência for terminada com *CHAR_MAX*, nenhum agrupamento adicional é realizado. Se a sequência termina com um 0, o tamanho do último grupo é usado repetidamente.
	'thousands_sep'	Caractere usado entre grupos.
LC_MONETARY	'int_curr_symbol'	Símbolo de moeda internacional.
	'currency_symbol'	Símbolo de moeda local.
	'p_cs_precedes/n_cs_precedes'	Se o símbolo da moeda precede o valor (para valores positivos ou negativos).
	'p_sep_by_space/n_sep_by_space'	Se o símbolo monetário está separado do valor por um espaço (para valores positivos ou negativos).
	'mon_decimal_point'	Ponto decimal usado para valores monetários.
	'frac_digits'	Número de dígitos fracionários usados na formatação local de valores monetários.
	'int_frac_digits'	Número de dígitos fracionários usados na formatação internacional de valores monetários.
	'mon_thousands_sep'	Separador de grupo usado para valores monetários.
	'mon_grouping'	Equivalente a 'grouping', usado para valores monetários.
	'positive_sign'	Símbolo usado para anotar um valor monetário positivo.
	'negative_sign'	Símbolo usado para anotar um valor monetário negativo.
	'p_sign_posn/n_sign_posn'	A posição do sinal (para valores positivos resp. negativos), veja abaixo.

Todos os valores numéricos podem ser definidos como *CHAR_MAX* para indicar que não há valor especificado nesta localidade.

Os valores possíveis para 'p_sign_posn' e 'n_sign_posn' são dados abaixo.

Valor	Explanação
0	A moeda e o valor estão entre parênteses.
1	O sinal deve preceder o valor e o símbolo da moeda.
2	O sinal deve seguir o valor e o símbolo da moeda.
3	O sinal deve preceder imediatamente o valor.
4	O sinal deve seguir imediatamente o valor.
CHAR_MAX	Nada é especificado nesta localidade.

A função define temporariamente a localidade de LC_CTYPE para a localidade de LC_NUMERIC ou a localidade de LC_MONETARY se as localidades forem diferentes e as strings numéricas ou monetárias não forem ASCII. Esta mudança temporária afeta outras threads.

Alterado na versão 3.7: A função agora define temporariamente a localidade de LC_CTYPE para a localidade de LC_NUMERIC em alguns casos.

locale.**nl_langinfo**(*option*)

Retorna algumas informações específicas da localidade em uma string. Esta função não está disponível em todos os sistemas e o conjunto de opções possíveis também pode variar entre as plataformas. Os valores de argumento possíveis são números, para os quais constantes simbólicas estão disponíveis no módulo da localidade.

A função *nl_langinfo()* aceita uma das seguintes chaves. A maioria das descrições são tiradas da descrição correspondente na biblioteca GNU C.

locale.**CODESET**
Obtém uma string com o nome da codificação de caracteres usada na localidade selecionado.

locale.**D_T_FMT**
Obtém uma string que pode ser usada como uma string de formato para *time.strftime()* para representar a data e a hora de uma maneira específica da localidade.

locale.**D_FMT**
Obtém uma string que pode ser usada como uma string de formato para *time.strftime()* para representar uma data de uma maneira específica da localidade.

locale.**T_FMT**
Obtém uma string que pode ser usada como uma string de formato para *time.strftime()* para representar uma hora de uma maneira específica da localidade.

locale.**T_FMT_AMPM**
Obtém uma string de formato para *time.strftime()* para representar a hora no formato am/pm.

DAY_1 ... DAY_7
Obtém o nome do enésimo dia da semana.

Nota: Isso segue a convenção dos EUA de DAY_1 ser no domingo, não a convenção internacional (ISO 8601) que segunda-feira é o primeiro dia da semana.

ABDAY_1 ... ABDAY_7
Obtém o nome abreviado do enésimo dia da semana.

MON_1 ... MON_12
Obtém o nome do enésimo dia do mês.

ABMON_1 ... ABMON_12
Obtém o nome abreviado do enésimo dia do mês.

locale.**RADIXCHAR**
Obtém o caractere separador (ponto decimal, vírgula decimal etc.).

locale.**THOUSEP**
Obtém o caractere separador para milhares (grupos de três dígitos).

locale.**YESEXPR**
Obtém uma expressão regular que pode ser usada com a função regex para reconhecer uma resposta positiva a uma pergunta sim/não.

Nota: A expressão está na sintaxe adequada para a função `regex()` da biblioteca C, que pode ser diferente da sintaxe usada em *re*.

`locale.`**NOEXPR**
Obtém uma expressão regular que pode ser usada com a função regex(3) para reconhecer uma resposta negativa a uma pergunta sim/não.

`locale.`**CRNCYSTR**
Obtém o símbolo da moeda, precedido por "-" se o símbolo deve aparecer antes do valor, "+" se o símbolo deve aparecer após o valor ou "." se o símbolo deve substituir o caractere separador.

`locale.`**ERA**
Obtém uma string que represente a era usada na localidade atual.

A maioria das localidades não define esse valor. Um exemplo de localidade que define esse valor é o japonês. No Japão, a representação tradicional de datas inclui o nome da época correspondente ao reinado do então imperador.

Normalmente não deve ser necessário usar este valor diretamente. Especificar o modificador `E` em suas strings de formato faz com que a função *time.strftime()* use esta informação. O formato da string retornada não é especificado e, portanto, você não deve presumir que tem conhecimento dele em sistemas diferentes.

`locale.`**ERA_D_T_FMT**
Obtém uma string de formato para *time.strftime()* para representar a data e a hora de uma forma baseada na era específica da localidade.

`locale.`**ERA_D_FMT**
Obtém uma string de formato para *time.strftime()* para representar uma data em uma forma baseada em era específica da localidade.

`locale.`**ERA_T_FMT**
Obtém uma string de formato para *time.strftime()* para representar uma hora em uma forma baseada em era específica da localidade.

`locale.`**ALT_DIGITS**
Obtém uma representação de até 100 valores usada para representar os valores de 0 a 99.

`locale.`**getdefaultlocale**([*envvars*])
Tenta determinar as configurações de localidade padrão e as retorna como uma tupla na forma `(language code, encoding)`.

De acordo com POSIX, um programa que não chamou `setlocale(LC_ALL, '')` executa usando a localidade portátil `'C'`. Chamar `setlocale(LC_ALL, '')` permite que ele use a localidade padrão conforme definido pela variável `LANG`. Como não queremos interferir com a configuração de localidade atual, emulamos o comportamento da maneira descrita acima.

Para manter a compatibilidade com outras plataformas, não apenas a variável `LANG` é testada, mas uma lista de variáveis fornecida como parâmetro envvars. Será utilizado o primeiro encontrado a ser definido. *envvars* padroniza para o caminho de pesquisa usado no GNU gettext; deve sempre conter o nome da variável `'LANG'`. O caminho de pesquisa do GNU gettext contém `'LC_ALL'`, `'LC_CTYPE'`, `'LANG'` e `'LANGUAGE'`, nesta ordem.

Exceto pelo código `'C'`, o código do idioma corresponde a **RFC 1766**. *language code* e *encoding* podem ser `None` se seus valores não puderem ser determinados.

`locale.`**getlocale**(*category=LC_CTYPE*)
Retorna a configuração atual para a categoria de localidade fornecida como uma sequência contendo *language code*, *encoding*. *category* pode ser um dos valores `LC_*`, exceto *LC_ALL*. O padrão é *LC_CTYPE*.

Exceto pelo código `'C'`, o código do idioma corresponde a **RFC 1766**. *language code* e *encoding* podem ser `None` se seus valores não puderem ser determinados.

`locale.`**`getpreferredencoding`**(*do_setlocale=True*)
Retorna a codificação usada para dados de texto, de acordo com as preferências do usuário. As preferências do usuário são expressas de maneira diferente em sistemas diferentes e podem não estar disponíveis programaticamente em alguns sistemas, portanto, essa função retorna apenas uma estimativa.

Em alguns sistemas, é necessário invocar *setlocale()* para obter as preferências do usuário, portanto, esta função não é segura para thread. Se invocar setlocale não for necessário ou desejado, *do_setlocale* deve ser definido como `False`.

No Android ou no modo UTF-8 (opção `-X utf8`), sempre retorna `'UTF-8'`, a localidade e o argumento *do_setlocale* são ignorados.

Alterado na versão 3.7: A função agora sempre retorna `UTF-8` no Android ou se o modo UTF-8 estiver habilitado.

`locale.`**`normalize`**(*localename*)
Retorna um código de localidade normalizado para o nome de localidade fornecido. O código de localidade retornado é formatado para uso com *setlocale()*. Se a normalização falhar, o nome original será retornado inalterado.

Se a codificação fornecida não for conhecida, o padrão da função é a codificação padrão para o código da localidade, assim como *setlocale()*.

`locale.`**`resetlocale`**(*category=LC_ALL*)
Define o localidade de *category* para a configuração padrão.

A configuração padrão é determinada chamando *getdefaultlocale()*. *category* tem como padrão *LC_ALL*.

`locale.`**`strcoll`**(*string1*, *string2*)
Compara duas strings de acordo com a configuração atual *LC_COLLATE*. Como qualquer outra função de comparação, retorna um valor negativo ou positivo, ou `0`, dependendo se *string1* agrupa antes ou depois de *string2* ou é igual a ele.

`locale.`**`strxfrm`**(*string*)
Transforma uma string em uma que pode ser usada em comparações com reconhecimento de localidade. Por exemplo, `strxfrm(s1) < strxfrm(s2)` é equivalente a `strcoll(s1, s2) < 0`. Esta função pode ser usada quando a mesma string é comparada repetidamente, por exemplo, ao agrupar uma sequência de strings.

`locale.`**`format_string`**(*format*, *val*, *grouping=False*, *monetary=False*)
Formata um número *val* de acordo com a configuração atual de *LC_NUMERIC*. O formato segue as convenções do operador `%`. Para valores de ponto flutuante, o ponto decimal é modificado, se apropriado. Se *grouping* for verdadeiro, também leva o agrupamento em consideração.

Se *monetary* for verdadeiro, a conversão usa o separador de milhares monetários e strings de agrupamento.

Processa especificadores de formatação como em `format % val`, mas leva as configurações de localidade atuais em consideração.

Alterado na versão 3.7: O parâmetro nomeado *monetary* foi adicionado.

`locale.`**`format`**(*format*, *val*, *grouping=False*, *monetary=False*)
Observe que esta função funciona como *format_string()*, mas só funcionará para exatamente um especificador `%char`. Por exemplo, `'%f'` e `'%.0f'` são especificadores válidos, mas `'%f KiB'` não é.

Para strings de formato inteiras, use *format_string()*.

Obsoleto desde a versão 3.7: Use *format_string()*.

`locale.`**`currency`**(*val*, *symbol=True*, *grouping=False*, *international=False*)
Formata um número *val* de acordo com as configurações atuais de *LC_MONETARY*.

A string retornada inclui o símbolo da moeda se *symbol* for verdadeiro, que é o padrão. Se *grouping* for verdadeiro (o que não é o padrão), o agrupamento é feito com o valor. Se *international* for verdadeiro (o que não é o padrão), o símbolo da moeda internacional será usado.

Observe que esta função não funcionará com a localidade 'C', então você deve definir uma localidade via *setlocale()* primeiro.

locale.**str**(*float*)
Formata um número de ponto flutuante usando o mesmo formato da função embutida `str(float)`, mas leva o ponto decimal em consideração.

locale.**delocalize**(*string*)
Converte uma string em uma string numérica normalizada, seguindo as configurações de *LC_NUMERIC*.

Novo na versão 3.5.

locale.**atof**(*string*)
Converte uma string em um número de ponto flutuante, seguindo as configurações de *LC_NUMERIC*.

locale.**atoi**(*string*)
Converte uma string em um número inteiro, seguindo as convenções de *LC_NUMERIC*.

locale.**LC_CTYPE**
Categoria da localidade para as funções de tipo de caractere. Dependendo das configurações desta categoria, as funções do módulo *string* lidando com diferença de maiúsculo e minúsculo mudam seu comportamento.

locale.**LC_COLLATE**
Categoria da localidade para classificação de strings. As funções *strcoll()* e *strxfrm()* do módulo *locale* são afetadas.

locale.**LC_TIME**
Categoria da localidade para a formatação de hora. A função *time.strftime()* segue essas convenções.

locale.**LC_MONETARY**
Categoria da localidade para formatação de valores monetários. As opções disponíveis estão disponíveis na função *localeconv()*.

locale.**LC_MESSAGES**
Categoria da localidade para exibição de mensagens. Python atualmente não oferece suporte a mensagens com reconhecimento de localidade específicas da aplicação. Mensagens exibidas pelo sistema operacional, como aquelas retornadas por *os.strerror()* podem ser afetadas por esta categoria.

locale.**LC_NUMERIC**
Categoria da localidade para formatação de números. As funções *format()*, *atoi()*, *atof()* e *str()* do módulo *locale* são afetadas por essa categoria. Todas as outras operações de formatação numérica não são afetadas.

locale.**LC_ALL**
Combinação de todas as configurações da localidade. Se este sinalizador for usado quando a localidade for alterada, a configuração da localidade para todas as categorias será tentada. Se isso falhar para qualquer categoria, nenhuma categoria é alterada. Quando a localidade é recuperada usando este sinalizador, uma string indicando a configuração para todas as categorias é retornada. Esta string pode ser usada posteriormente para restaurar as configurações.

locale.**CHAR_MAX**
Esta é uma constante simbólica usada para diferentes valores retornados por *localeconv()*.

Exemplo:

```
>>> import locale
>>> loc = locale.getlocale()  # get current locale
# use German locale; name might vary with platform
```

(continua na próxima página)

(continuação da página anterior)

```
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('f\xe4n', 'foo')  # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '')    # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C')   # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc)   # restore saved locale
```

24.2.1 Histórico, detalhes, dicas, dicas e advertências

O padrão C define a localidade como uma propriedade de todo o programa que pode ser relativamente cara para alterar. Além disso, algumas implementações são interrompidas de forma que mudanças frequentes de localidade podem causar despejos de memória. Isso torna a localidade um tanto dolorosa de usar corretamente.

Inicialmente, quando um programa é iniciado, a localidade é a localidade `C`, não importa qual a localidade preferida do usuário. Há uma exceção: a categoria *LC_CTYPE* é alterada na inicialização para definir a codificação de localidade atual para a codificação de localidade preferida do usuário. O programa deve dizer explicitamente que deseja as configurações de localidade preferidas do usuário para outras categorias chamando `setlocale(LC_ALL, '')`.

Geralmente é uma má ideia chamar *setlocale()* em alguma rotina de biblioteca, já que como efeito colateral afeta todo o programa. Salvar e restaurar é quase tão ruim: é caro e afeta outras threads que são executadas antes de as configurações serem restauradas.

Se, ao codificar um módulo para uso geral, você precisa de uma versão independente da localidade de uma operação que é afetada pela localidade (como certos formatos usados com *time.strftime()*), você terá que encontrar uma maneira de faça isso sem usar a rotina de biblioteca padrão. Melhor ainda é se convencer de que não há problema em usar as configurações da localidade. Apenas como último recurso, você deve documentar que seu módulo não é compatível com configurações de localidade não-`C`.

A única maneira de realizar operações numéricas de acordo com a localidade é usar as funções especiais definidas por este módulo: *atof()*, *atoi()*, *format()*, *str()*.

Não há como realizar conversões de maiúsculas e minúsculas e classificações de caracteres de acordo com a localidade. Para strings de texto (Unicode), isso é feito de acordo com o valor do caractere apenas, enquanto para strings de byte, as conversões e classificações são feitas de acordo com o valor ASCII do byte e bytes cujo bit alto está definido (ou seja, bytes não ASCII) nunca são convertidos ou considerados parte de uma classe de caracteres, como letras ou espaços em branco.

24.2.2 Para escritores de extensão e programas que incorporam Python

Módulos de extensão nunca devem chamar *setlocale()*, exceto para descobrir qual é a localidade atual. Mas uma vez que o valor de retorno só pode ser usado portavelmente para restaurá-lo, isso não é muito útil (exceto talvez para descobrir se a localidade é ou não `C`).

Quando o código Python usa o módulo *locale* para alterar a localidade, isso também afeta a aplicação de incorporação. Se a aplicação de incorporação não quiser que isso aconteça, ele deve remover o módulo de extensão `_locale` (que faz todo o trabalho) da tabela de módulos embutidos no arquivo `config.c` e certificar-se de que o módulo `_locale` não está acessível como uma biblioteca compartilhada.

24.2.3 Acesso a catálogos de mensagens

`locale.`**`gettext`**(*msg*)

`locale.`**`dgettext`**(*domain*, *msg*)

`locale.`**`dcgettext`**(*domain*, *msg*, *category*)

`locale.`**`textdomain`**(*domain*)

`locale.`**`bindtextdomain`**(*domain*, *dir*)

O módulo locale expõe a interface gettext da biblioteca C em sistemas que fornecem essa interface. Consiste nas funções `gettext()`, `dgettext()`, `dcgettext()`, `textdomain()`, `bindtextdomain()` e `bind_textdomain_codeset()`. Elas são semelhantes às mesmas funções no módulo *gettext*, mas usam o formato binário da biblioteca C para catálogos de mensagens e os algoritmos de pesquisa da biblioteca C para localizar catálogos de mensagens.

As aplicações Python normalmente não precisam invocar essas funções e devem usar *gettext* em seu lugar. Uma exceção conhecida a esta regra são os aplicativos que se vinculam a bibliotecas C adicionais que invocam internamente `gettext()` ou `dcgettext()`. Para essas aplicações, pode ser necessário vincular o domínio de texto, para que as bibliotecas possam localizar adequadamente seus catálogos de mensagens.

CAPÍTULO 25

Frameworks de programa

Os módulos descritos neste capítulo são frameworks que ditarão em grande parte a estrutura do seu programa. Atualmente, os módulos descritos aqui são todos orientados para escrever interfaces de linha de comando.

A lista completa de módulos descritos neste capítulo é:

25.1 `turtle` — Gráficos Turtle

Código Fonte: Lib/turtle.py

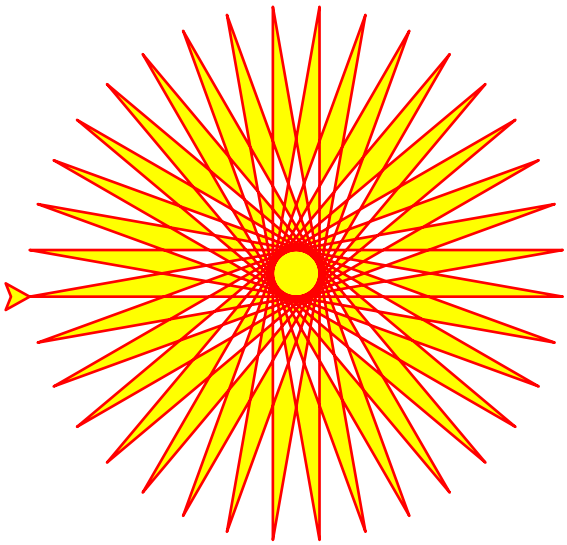
25.1.1 Introdução

Gráficos turtle é uma forma popular de introduzir programação para crianças. Era parte da linguagem de programção Logo desenvolvida por Wally Feuzeig, Seymour Papert and Cynthia Solomon em 1967.

Imagine uma tartaruga robótica começando em (0, 0) no plano x-y. Depois de um "import turtle", dê-lhe o comando `turtle.forward(15)`, e mova (na tela!) 15 pixels na direção em que está virada, desenhando uma linha à medida que ela se move. Digite o comando `turtle.right(25)`, e gire-a no lugar 25 graus no sentido horário.

Turtle star

A tartaruga pode desenhar formas intrincadas usando programas que repetem movimentos simples.



```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

Ao combinar esses comandos similares, formas intrincadas e imagens podem ser desenhadas facilmente.

O módulo *turtle* é uma re-implementação extendida de um módulo de mesmo nome da distribuição padrão do Python até a versão Python 2.5.

Ele tenta manter os valores do antigo módulo turtle e ser (quase) 100% compatível com ele. Isso significa, em primeiro lugar, permitir que o programador aprendiz use todos os comandos, classes e métodos interativamente ao usar o módulo de dentro do IDLE executado com a chave `-n`.

O módulo *turtle* fornece gráficos rudimentares, tanto para programação orientada a objetos quanto procedural. Como ele usa o *tkinter* como módulo gráfico, ele necessita de uma versão do instalada do Python que suporta o Tk.

A interface orientada a objetos usa essencialmente duas classes (e duas sub-classes):

1. A classe *TurtleScreen* define as janelas gráficas como um parque de diversões para as tartarugas de desenho. Seu construtor precisa de um `tkinter.Canvas` ou um *ScrolledCanvas* como argumento. Deve ser usado quando *turtle* é usado como parte de algum aplicativo.

   A função *Screen()* retorna um objeto singleton de uma subclasse *TurtleScreen*. Esta função deve ser usada quando *turtle* é usado como uma ferramenta autônoma para fazer gráficos. Como um objeto singleton, não é possível herdar de sua classe.

   Todos os métodos de TurtleScreen/Screen também existem como funções, ou seja, como parte da interface orientada a procedimentos.

2. *RawTurtle* (pedido: *RawPen*) define objetos Turtle que desenham em uma *TurtleScreen*. Seu construtor precisa de um Canvas, ScrolledCanvas ou TurtleScreen como argumento, para que os objetos RawTurtle saibam

onde desenhar.

Derivada de RawTurtle é a subclasse *Turtle* (alias: Pen), que se baseia "na" instância *Screen* que é criada automaticamente, se ainda não estiver presente.

Todos os métodos de RawTurtle/Turtle também existem como funções, isto é, parte de uma interface procedural orientada à objeto.

A interface procedural fornece funções que são derivadas dos métodos das classes *Screen* e *Turtle*. Eles têm os mesmos nomes que os métodos correspondentes. Um objeto de tela é criado automaticamente sempre que uma função derivada de um método de tela é chamada. Um objeto turtle (sem nome) é criado automaticamente sempre que qualquer uma das funções derivadas de um método Turtle é chamada.

Para usar várias tartarugas em uma tela, é preciso usar a interface orientada a objetos.

Nota: Na documentação a seguir, a lista de argumentos para funções é fornecida. Os métodos, é claro, têm o primeiro argumento adicional *self* que é omitido aqui.

25.1.2 Visão geral dos métodos Turtle e Screen disponíveis

Métodos do Turtle

Movimentos do Turtle

Movimento e Desenho

forward() | *fd()*
backward() | *bk()* | *back()*
right() | *rt()*
left() | *lt()*
goto() | *setpos()* | *setposition()*
setx()
sety()
setheading() | *seth()*
home()
circle()
dot()
stamp()
clearstamp()
clearstamps()
undo()
speed()

Fala o Estado da Tartaruga

position() | *pos()*
towards()
xcor()
ycor()
heading()
distance()

Configuração e Medidas

degrees()
radians()

Controle da Caneta

Estado do Desenho

pendown() | *pd()* | *down()*
penup() | *pu()* | *up()*
pensize() | *width()*
pen()
isdown()

Controle da Cor

color()
pencolor()
fillcolor()

Preenchimento

filling()
begin_fill()
end_fill()

Mais sobre o Controle do Desenho

reset()
clear()
write()

Estado da Tartaruga

Visibilidade

showturtle() | *st()*
hideturtle() | *ht()*
isvisible()

Aparência

shape()
resizemode()
shapesize() | *turtlesize()*
shearfactor()
settiltangle()
tiltangle()
tilt()
shapetransform()
get_shapepoly()

Eventos Utilizados

onclick()
onrelease()
ondrag()

Métodos Especiais da Tartaruga

begin_poly()
end_poly()
get_poly()
clone()
getturtle() | *getpen()*
getscreen()
setundobuffer()
undobufferentries()

Métodos de TurtleScreen/Screen

Controle da Janela

bgcolor()
bgpic()
clear() | *clearscreen()*
reset() | *resetscreen()*
screensize()
setworldcoordinates()

Controle da Animação

delay()
tracer()
update()

Usando os Eventos de Tela

listen()
onkey() | *onkeyrelease()*
onkeypress()
onclick() | *onscreenclick()*
ontimer()
mainloop() | *done()*

Configurações e Métodos Especiais

mode()
colormode()
getcanvas()
getshapes()
register_shape() | *addshape()*
turtles()
window_height()
window_width()

Métodos de Entrada

textinput()
numinput()

Métodos Específicos para Tela

bye()

exitonclick()
setup()
title()

25.1.3 Métodos de RawTurtle/Turtle e funções correspondentes

A maioria dos exemplos desta seção referem-se a uma ocorrência Tartaruga chamada `turtle`.

Movimentos do Turtle

`turtle.`**`forward`**(*distance*)
`turtle.`**`fd`**(*distance*)

Parâmetros **`distance`** – a number (integer or float)

Move a tartaruga para frente pela *distance* especificada, na direção em que a tartaruga está indo.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

`turtle.`**`back`**(*distance*)
`turtle.`**`bk`**(*distance*)
`turtle.`**`backward`**(*distance*)

Parâmetros **`distance`** – um número

Move a tartaruga para trás por *distance*, na direção oposta à direção em que a tartaruga está indo. Não muda o rumo da tartaruga.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

`turtle.`**`right`**(*angle*)
`turtle.`**`rt`**(*angle*)

Parâmetros **`angle`** – a number (integer or float)

Vira a tartaruga à direita por unidades de *angle*. (As unidades são por padrão graus, mas podem ser definidas através das funções *degrees()* e *radians()*.) A orientação do ângulo depende do modo tartaruga, veja *mode()*.

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

`turtle.`**`left`**(*angle*)
`turtle.`**`lt`**(*angle*)

Parâmetros **angle** – a number (integer or float)

Vira a tartaruga à esquerda por unidades de *angle*. (As unidades são por padrão graus, mas podem ser definidas através das funções *degrees()* e *radians()*.) A orientação do ângulo depende do modo tartaruga, veja *mode()*.

```
>>> turtle.heading()
22.0
>>> turtle.left(45)
>>> turtle.heading()
67.0
```

turtle.**goto**(*x*, *y=None*)
turtle.**setpos**(*x*, *y=None*)
turtle.**setposition**(*x*, *y=None*)

Parâmetros

- **x** – um número ou um par/arrays de números
- **y** – um número ou None

Se *y* for None, *x* deve ser um par de coordenadas ou uma classe *Vec2D* (por exemplo, como retornado pela função *pos()*).

Move a tartaruga para uma posição absoluta. Caso a caneta esteja baixa, trace a linha. Não altera a orientação da tartaruga.

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00,0.00)
```

turtle.**setx**(*x*)

Parâmetros **x** – a number (integer or float)

Define a primeira coordenada da tartaruga para *x*, deixa a segunda coordenada inalterada.

```
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

turtle.**sety**(*y*)

Parâmetros **y** – a number (integer or float)

Defina a segunda coordenada da tartaruga para *y*, deixa a primeira coordenada inalterada.

```
>>> turtle.position()
(0.00,40.00)
```

(continua na próxima página)

(continuação da página anterior)

```
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

turtle.**setheading**(*to_angle*)
turtle.**seth**(*to_angle*)

Parâmetros **to_angle** – a number (integer or float)

Determine a orientação da tartaruga para *to_angle*. Aqui estão algumas direções mais comuns em graus:

modo padrão	logo mode
0 - leste	0 - norte
90 - norte	90 - leste
180 - oeste	180 - sul
270 - sul	270 - oeste

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

turtle.**home**()

Move a tartaruga para a origem – coordenadas (0,0) – e define seu rumo para sua orientação inicial (que depende do modo, veja *mode()*).

```
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**circle**(*radius*, *extent=None*, *steps=None*)

Parâmetros

- **radius** – um número
- **extent** – um número (ou None)
- **steps** – an integer (or None)

Desenha um círculo com dado *radius*. O centro são as unidades de *radius* à esquerda da tartaruga; *extent* – um ângulo – determina qual parte do círculo é desenhada. Se *extent* não for fornecida, desenha o círculo inteiro. Se *extent* não for um círculo completo, uma extremidade do arco será a posição atual da caneta. Desenha o arco no sentido anti-horário se *radius* for positivo, caso contrário, no sentido horário. Finalmente, a direção da tartaruga é alterada pela quantidade de *extent*.

Como o círculo é aproximado por um polígono regular inscrito, *steps* determina o número de passos a serem usados. Caso não seja informado, será calculado automaticamente. Pode ser usado para desenhar polígonos regulares.

```
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
```

(continua na próxima página)

(continuação da página anterior)

```
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180)  # draw a semicircle
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0
```

turtle.**dot**(*size=None*, **color*)

Parâmetros

- **size** – um número >= 1 (caso seja fornecido)
- **color** – uma String de cores ou uma tupla de cores numéricas

Desenha um ponto circular com diâmetro *size*, usando *color*. Se *size* não for fornecido, o máximo de pensize+4 e 2*pensize será usado.

```
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0
```

turtle.**stamp**()

Carimba uma cópia da forma da tartaruga na tela na posição atual da tartaruga. Retorna um stamp_id para esse carimbo, que pode ser usado para excluí-lo chamando `clearstamp(stamp_id)`.

```
>>> turtle.color("blue")
>>> turtle.stamp()
11
>>> turtle.fd(50)
```

turtle.**clearstamp**(*stampid*)

Parâmetros **stampid** – um inteiro, deve ser o valor de retorno da chamada de *stamp()* anterior

Exclui o carimbo com o *stamp* fornecido.

```
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00,-0.00)
```

turtle.**clearstamps**(*n=None*)

Parâmetros **n** – an integer (or None)

Exclui todos ou o primeiro/último *n* dos selos da tartaruga. Se *n* for None, exclui todos os carimbos, se $n > 0$ exclui os primeiros *n* carimbos, senão se $n < 0$ exclui os últimos *n* carimbos.

```
>>> for i in range(8):
...     turtle.stamp(); turtle.fd(30)
13
14
15
16
17
18
19
20
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

turtle.**undo**()

Desfaz (repetidamente) a(s) última(s) ação(ões) da tartaruga. O número de ações de desfazer disponíveis é determinado pelo tamanho do buffer de desfazer.

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

turtle.**speed**(*speed=None*)

Parâmetros **speed** – um inteiro no intervalo 0..10 ou uma string de velocidade (veja abaixo)

Define a velocidade da tartaruga para um valor inteiro no intervalo 0..10. Se nenhum argumento for fornecido, retorna a velocidade atual.

Se a entrada for um número maior que 10 ou menor que 0,5, a velocidade é definida como 0. As strings de velocidade são mapeadas para valores de velocidade da seguinte forma:

- "fastest": 0
- "fast": 10
- "normal": 6
- "slow": 3
- "slowest": 1

Velocidades de 1 a 10 tornam a animação cada vez mais rápida, tanto para o desenho da linha como para a rotação da tartaruga.

Atenção: *speed* = 0 significa que *nenhuma* animação ocorre. Para frente/trás faz a tartaruga pular e da mesma forma para esquerda/direita faz a tartaruga girar instantaneamente.

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
```

(continua na próxima página)

(continuação da página anterior)

```
>>> turtle.speed()
9
```

Fala o Estado da Tartaruga

turtle.**position**()
turtle.**pos**()
Retorna a localização atual da tartaruga (x,y) (como um vetor *Vec2D*).

```
>>> turtle.pos()
(440.00,-0.00)
```

turtle.**towards**(*x*, *y=None*)

Parâmetros

- **x** – um número ou um par/vetor de números ou uma instância de tartaruga
- **y** – um número caso *x* seja um número, senão None

Return the angle between the line from turtle position to position specified by (x,y), the vector or the other turtle. This depends on the turtle's start orientation which depends on the mode - "standard"/"world" or "logo").

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

turtle.**xcor**()
Retorna a coordenada X da tartaruga.

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28,76.60)
>>> print(round(turtle.xcor(), 5))
64.27876
```

turtle.**ycor**()
Retorna a coordenada Y da tartaruga

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00,86.60)
>>> print(round(turtle.ycor(), 5))
86.60254
```

turtle.**heading**()
Retorna o título atual da tartaruga (o valor depende do modo da tartaruga, veja *mode()*).

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

turtle.**distance**(*x*, *y=None*)

Parâmetros

- **x** – um número ou um par/vetor de números ou uma instância de tartaruga
- **y** – um número caso *x* seja um número, senão `None`

Retorna a distância da tartaruga para (x,y), o vetor dado, ou a outra tartaruga dada, em unidades de passo de tartaruga.

```
>>> turtle.home()
>>> turtle.distance(30,40)
50.0
>>> turtle.distance((30,40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

Configurações de Medida

turtle.**degrees**(*fullcircle=360.0*)

Parâmetros **fullcircle** – um número

Define as unidades de medição do ângulo, ou seja, defina o número de "graus" para um círculo completo. O valor padrão é 360 graus.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

Change angle measurement unit to grad (also known as gon,
grade, or gradian and equals 1/100-th of the right angle.)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

turtle.**radians**()

Define as unidades de medida de ângulo para radianos. Equivalente a `degrees(2*math.pi)`.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

Controle da Caneta

Estado do Desenho

turtle.**pendown**()
turtle.**pd**()
turtle.**down**()
Desce a caneta - desenha ao se mover.

turtle.**penup**()
turtle.**pu**()
turtle.**up**()
Levanta a caneta – sem qualquer desenho ao se mover.

turtle.**pensize**(*width=None*)
turtle.**width**(*width=None*)

Parâmetros **width** – um número positivo

Define a espessura da linha para *width* ou retorne-a. Se resizemode estiver definido como "auto" e a forma de tartaruga for um polígono, esse polígono será desenhado com a mesma espessura de linha. Se nenhum argumento for fornecido, o tamanho da pena atual será retornado.

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)   # from here on lines of width 10 are drawn
```

turtle.**pen**(*pen=None*, ***pendict*)

Parâmetros

- **pen** – um dicionário com algumas ou todas as chaves listadas abaixo
- **pendict** – um ou mais argumentos nomeados com as chaves listadas abaixo como palavras-chave

Retorna ou define os atributos da caneta em um "dicionário da caneta" com os seguintes pares de chave/valor:

- "shown": True/False
- "pendown": True/False
- "pencolor": color-string or color-tuple
- "fillcolor": color-string or color-tuple
- "pensize": positive number
- "speed": number in range 0..10
- "resizemode": "auto" or "user" or "noresize"
- "stretchfactor": (positive number, positive number)
- "outline": positive number
- "tilt": number

Este dicionário pode ser usado como argumento para uma chamada subsequente para `pen()` para restaurar o estado da caneta anterior. Além disso, um ou mais desses atributos podem ser fornecidos como argumentos nomeados. Isso pode ser usado para definir vários atributos de caneta em uma instrução.

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shearfactor', 0.0), ('shown', True), ('speed', 9),
 ('stretchfactor', (1.0, 1.0)), ('tilt', 0.0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow')]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red')]
```

`turtle.`**`isdown`**`()`

Retorna `True` se a caneta estiver baixa, `False` se estiver feito.

```
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```

Controle da Cor

`turtle.`**`pencolor`**`(*args)`

Retorna ou define o pencolor.

São permitidos quatro formatos de entrada:

`pencolor()` Retorna a cor da caneta atual como string de especificação de cor ou como uma tupla (veja o exemplo). Pode ser usado como entrada para outra chamada color/pencolor/fillcolor.

`pencolor(colorstring)` Define pencolor como *colorstring*, que é uma string de especificação de cor Tk, como `"red"`, `"yellow"` ou `"#33cc8c"`.

`pencolor((r, g, b))` Define a cor da caneta como a cor RGB representada pela tupla *r*, *g*, e *b*. Os valores de *r*, *g*, and *b* precisam estar na faixa 0..colormode, onde colormode é 1.0 ou 255 (ver *`colormode()`*).

`pencolor(r, g, b)`

Define a cor da caneta como a cor RGB representada por *r*, *g*, e *b*. Os valores de *r*, *g*, and *b* precisam estar na faixa 0..colormode, onde colormode é 1.0 ou 255 (ver `colormode()`).

Se a forma da tartaruga for um polígono, o contorno desse polígono será desenhado com a nova cor de caneta definida.

```
>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
```

(continua na próxima página)

(continuação da página anterior)

```
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
>>> colormode(255)
>>> turtle.pencolor()
(51.0, 204.0, 140.0)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

turtle.**fillcolor**(**args*)

Retorna ou define o fillcolor.

São permitidos quatro formatos de entrada:

fillcolor() Return the current fillcolor as color specification string, possibly in tuple format (see example). May be used as input to another color/pencolor/fillcolor call.

fillcolor(colorstring) Set fillcolor to *colorstring*, which is a Tk color specification string, such as `"red"`, `"yellow"`, or `"#33cc8c"`.

fillcolor((r, g, b)) Set fillcolor to the RGB color represented by the tuple of *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode, where colormode is either 1.0 or 255 (see *colormode()*).

fillcolor(r, g, b)

Set fillcolor to the RGB color represented by *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode.

If turtleshape is a polygon, the interior of that polygon is drawn with the newly set fillcolor.

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor((50, 193, 143))  # Integers, not floats
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255.0, 255.0, 255.0)
```

turtle.**color**(**args*)

Return or set pencolor and fillcolor.

Several input formats are allowed. They use 0 to 3 arguments as follows:

color() Return the current pencolor and the current fillcolor as a pair of color specification strings or tuples as returned by *pencolor()* and *fillcolor()*.

color(colorstring), color((r,g,b)), color(r,g,b) Inputs as in *pencolor()*, set both, fillcolor and pencolor, to the given value.

color(colorstring1, colorstring2), color((r1,g1,b1), (r2,g2,b2))

Equivalent to `pencolor(colorstring1)` and `fillcolor(colorstring2)` and analogously if the other input format is used.

If turtleshape is a polygon, outline and interior of that polygon is drawn with the newly set colors.

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
>>> color("#285078", "#a0c8f0")
>>> color()
((40.0, 80.0, 120.0), (160.0, 200.0, 240.0))
```

Veja também: Método da tela *colormode()*.

Preenchimento

turtle.**filling**()
Retorna fillstate (`True` se estiver preenchido, `False` caso contrário).

```
>>> turtle.begin_fill()
>>> if turtle.filling():
...     turtle.pensize(5)
... else:
...     turtle.pensize(3)
```

turtle.**begin_fill**()
To be called just before drawing a shape to be filled.

turtle.**end_fill**()
Fill the shape drawn after the last call to *begin_fill()*.

Whether or not overlap regions for self-intersecting polygons or multiple shapes are filled depends on the operating system graphics, type of overlap, and number of overlaps. For example, the Turtle star above may be either all yellow or have some white regions.

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

Mais sobre o Controle do Desenho

turtle.**reset**()
Delete the turtle's drawings from the screen, re-center the turtle and set variables to the default values.

```
>>> turtle.goto(0,-22)
>>> turtle.left(100)
>>> turtle.position()
(0.00,-22.00)
>>> turtle.heading()
100.0
>>> turtle.reset()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**clear**()
Delete the turtle's drawings from the screen. Do not move turtle. State and position of the turtle as well as drawings of other turtles are not affected.

turtle.**write**(*arg*, *move=False*, *align="left"*, *font=("Arial", 8, "normal")*)

Parâmetros

- **arg** – object to be written to the TurtleScreen
- **move** – True/False
- **align** – uma das Strings "left", "center" ou right"
- **font** – a triple (fontname, fontsize, fonttype)

Write text - the string representation of *arg* - at the current turtle position according to *align* ("left", "center" or right") and with the given font. If *move* is true, the pen is moved to the bottom-right corner of the text. By default, *move* is `False`.

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

Estado da Tartaruga

Visibilidade

turtle.**hideturtle**()
turtle.**ht**()

Make the turtle invisible. It's a good idea to do this while you're in the middle of doing some complex drawing, because hiding the turtle speeds up the drawing observably.

```
>>> turtle.hideturtle()
```

turtle.**showturtle**()
turtle.**st**()

Tornar a tartaruga visível.

```
>>> turtle.showturtle()
```

turtle.**isvisible**()

Return `True` if the Turtle is shown, `False` if it's hidden.

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

Aparência

turtle.**shape**(*name=None*)

Parâmetros **name** – a string which is a valid shapename

Set turtle shape to shape with given *name* or, if name is not given, return name of current shape. Shape with *name* must exist in the TurtleScreen's shape dictionary. Initially there are the following polygon shapes: "arrow", "turtle", "circle", "square", "triangle", "classic". To learn about how to deal with shapes see Screen method *register_shape()*.

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

turtle.**resizemode**(*rmode=None*)

Parâmetros **rmode** – uma das Strings “auto”, “user”, “noresize”

Set resizemode to one of the values: “auto”, “user”, “noresize”. If *rmode* is not given, return current resizemode. Different resizemodes have the following effects:

- “auto”: adapta a aparência da tartaruga correspondente ao valor do pensize.
- “user”: adapts the appearance of the turtle according to the values of stretchfactor and outlinewidth (outline), which are set by *shapesize()*.
- “noresize”: no adaption of the turtle’s appearance takes place.

resizemode(“user”) is called by *shapesize()* when used with arguments.

```
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

turtle.**shapesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)
turtle.**turtlesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)

Parâmetros

- **stretch_wid** – número positivo
- **stretch_len** – número positivo
- **outline** – número positivo

Return or set the pen’s attributes x/y-stretchfactors and/or outline. Set resizemode to “user”. If and only if resizemode is set to “user”, the turtle will be displayed stretched according to its stretchfactors: *stretch_wid* is stretchfactor perpendicular to its orientation, *stretch_len* is stretchfactor in direction of its orientation, *outline* determines the width of the shapes’s outline.

```
>>> turtle.shapesize()
(1.0, 1.0, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

turtle.**shearfactor**(*shear=None*)

Parâmetros **shear** – número (optional)

Set or return the current shearfactor. Shear the turtleshape according to the given shearfactor shear, which is the tangent of the shear angle. Do *not* change the turtle’s heading (direction of movement). If shear is not given: return the current shearfactor, i. e. the tangent of the shear angle, by which lines parallel to the heading of the turtle are sheared.

```
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

turtle.**tilt**(*angle*)

Parâmetros **angle** – um número

Rotate the turtleshape by *angle* from its current tilt-angle, but do *not* change the turtle's heading (direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

turtle.**settiltangle**(*angle*)

Parâmetros **angle** – um número

Rotate the turtleshape to point in the direction specified by *angle*, regardless of its current tilt-angle. *Do not* change the turtle's heading (direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.settiltangle(45)
>>> turtle.fd(50)
>>> turtle.settiltangle(-45)
>>> turtle.fd(50)
```

Obsoleto desde a versão 3.1.

turtle.**tiltangle**(*angle=None*)

Parâmetros **angle** – um número (optional)

Set or return the current tilt-angle. If angle is given, rotate the turtleshape to point in the direction specified by angle, regardless of its current tilt-angle. Do *not* change the turtle's heading (direction of movement). If angle is not given: return the current tilt-angle, i. e. the angle between the orientation of the turtleshape and the heading of the turtle (its direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

turtle.**shapetransform**(*t11=None*, *t12=None*, *t21=None*, *t22=None*)

Parâmetros

- **t11** – um número (optional)
- **t12** – um número (optional)

- **t21** – um número (optional)
- **t12** – um número (optional)

Set or return the current transformation matrix of the turtle shape.

If none of the matrix elements are given, return the transformation matrix as a tuple of 4 elements. Otherwise set the given elements and transform the turtleshape according to the matrix consisting of first row t11, t12 and second row t21, 22. The determinant t11 * t22 - t12 * t21 must not be zero, otherwise an error is raised. Modify stretchfactor, shearfactor and tiltangle according to the given matrix.

```
>>> turtle = Turtle()
>>> turtle.shape("square")
>>> turtle.shapesize(4,2)
>>> turtle.shearfactor(-0.5)
>>> turtle.shapetransform()
(4.0, -1.0, -0.0, 2.0)
```

turtle.**get_shapepoly**()

Return the current shape polygon as tuple of coordinate pairs. This can be used to define a new shape or components of a compound shape.

```
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```

Eventos Utilizados

turtle.**onclick**(*fun*, *btn=1*, *add=None*)

Parâmetros

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – `True` or `False` – if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-click events on this turtle. If *fun* is `None`, existing bindings are removed. Example for the anonymous turtle, i.e. the procedural way:

```
>>> def turn(x, y):
...     left(180)
...
>>> onclick(turn)  # Now clicking into the turtle will turn it.
>>> onclick(None)  # event-binding will be removed
```

turtle.**onrelease**(*fun*, *btn=1*, *add=None*)

Parâmetros

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – `True` or `False` – if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-button-release events on this turtle. If *fun* is `None`, existing bindings are removed.

```
>>> class MyTurtle(Turtle):
...     def glow(self,x,y):
...         self.fillcolor("red")
...     def unglow(self,x,y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)      # clicking on turtle turns fillcolor red,
>>> turtle.onrelease(turtle.unglow)  # releasing turns it to transparent.
```

`turtle.`**`ondrag`**(*fun*, *btn=1*, *add=None*)

Parâmetros

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – `True` or `False` – if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-move events on this turtle. If *fun* is `None`, existing bindings are removed.

Remark: Every sequence of mouse-move-events on a turtle is preceded by a mouse-click event on that turtle.

```
>>> turtle.ondrag(turtle.goto)
```

Subsequently, clicking and dragging the Turtle will move it across the screen thereby producing handdrawings (if pen is down).

Métodos Especiais da Tartaruga

`turtle.`**`begin_poly`**()
Start recording the vertices of a polygon. Current turtle position is first vertex of polygon.

`turtle.`**`end_poly`**()
Stop recording the vertices of a polygon. Current turtle position is last vertex of polygon. This will be connected with the first vertex.

`turtle.`**`get_poly`**()
Return the last recorded polygon.

```
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

`turtle.`**`clone`**()
Create and return a clone of the turtle with same position, heading and turtle properties.

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

turtle.**getturtle**()
turtle.**getpen**()
Return the Turtle object itself. Only reasonable use: as a function to return the "anonymous turtle":

```
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

turtle.**getscreen**()
Return the *TurtleScreen* object the turtle is drawing on. TurtleScreen methods can then be called for that object.

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

turtle.**setundobuffer**(*size*)

Parâmetros **size** – um inteiro ou None

Set or disable undobuffer. If *size* is an integer an empty undobuffer of given size is installed. *size* gives the maximum number of turtle actions that can be undone by the *undo()* method/function. If *size* is None, the undobuffer is disabled.

```
>>> turtle.setundobuffer(42)
```

turtle.**undobufferentries**()
Return number of entries in the undobuffer.

```
>>> while undobufferentries():
...     undo()
```

Formas compostas

To use compound turtle shapes, which consist of several polygons of different color, you must use the helper class *Shape* explicitly as described below:

1. Create an empty Shape object of type "compound".
2. Add as many components to this object as desired, using the addcomponent() method.

   Por exemplo:

```
>>> s = Shape("compound")
>>> poly1 = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s.addcomponent(poly1, "red", "blue")
>>> poly2 = ((0,0),(10,-5),(-10,-5))
>>> s.addcomponent(poly2, "blue", "red")
```

3. Now add the Shape to the Screen's shapelist and use it:

```
>>> register_shape("myshape", s)
>>> shape("myshape")
```

Nota: The *Shape* class is used internally by the *register_shape()* method in different ways. The application programmer has to deal with the Shape class *only* when using compound shapes like shown above!

25.1.4 Methods of TurtleScreen/Screen and corresponding functions

Most of the examples in this section refer to a TurtleScreen instance called `screen`.

Controle da Janela

`turtle.`**`bgcolor`**(**args*)

Parâmetros **`args`** – a color string or three numbers in the range 0..colormode or a 3-tuple of such numbers

Set or return background color of the TurtleScreen.

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128.0, 0.0, 128.0)
```

`turtle.`**`bgpic`**(*picname=None*)

Parâmetros **`picname`** – a string, name of a gif-file or `"nopic"`, or `None`

Set background image or return name of current backgroundimage. If *picname* is a filename, set the corresponding image as background. If *picname* is `"nopic"`, delete background image, if present. If *picname* is `None`, return the filename of the current backgroundimage.

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

`turtle.`**`clear`**()
`turtle.`**`clearscreen`**()
Delete all drawings and all turtles from the TurtleScreen. Reset the now empty TurtleScreen to its initial state: white background, no background image, no event bindings and tracing on.

Nota: This TurtleScreen method is available as a global function only under the name `clearscreen`. The global function `clear` is a different one derived from the Turtle method `clear`.

`turtle.`**`reset`**()
`turtle.`**`resetscreen`**()
Reset all Turtles on the Screen to their initial state.

Nota: This TurtleScreen method is available as a global function only under the name `resetscreen`. The global function `reset` is another one derived from the Turtle method `reset`.

`turtle.`**`screensize`**(*canvwidth=None*, *canvheight=None*, *bg=None*)

Parâmetros

- **`canvwidth`** – positive integer, new width of canvas in pixels
- **`canvheight`** – positive integer, new height of canvas in pixels
- **`bg`** – colorstring or color-tuple, new background color

If no arguments are given, return current (canvaswidth, canvasheight). Else resize the canvas the turtles are drawing on. Do not alter the drawing window. To observe hidden parts of the canvas, use the scrollbars. With this method, one can make visible those parts of a drawing which were outside the canvas before.

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```

e.g. to search for an erroneously escaped turtle ;-)

`turtle.`**`setworldcoordinates`**(*llx*, *lly*, *urx*, *ury*)

Parâmetros

- **`llx`** – a number, x-coordinate of lower left corner of canvas
- **`lly`** – a number, y-coordinate of lower left corner of canvas
- **`urx`** – a number, x-coordinate of upper right corner of canvas
- **`ury`** – a number, y-coordinate of upper right corner of canvas

Set up user-defined coordinate system and switch to mode “world” if necessary. This performs a `screen.reset()`. If mode “world” is already active, all drawings are redrawn according to the new coordinates.

ATTENTION: in user-defined coordinate systems angles may appear distorted.

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
...
>>> for _ in range(8):
...     left(45); fd(2)   # a regular octagon
```

Controle da Animação

turtle.**delay**(*delay=None*)

Parâmetros **delay** – positive integer

Set or return the drawing *delay* in milliseconds. (This is approximately the time interval between two consecutive canvas updates.) The longer the drawing delay, the slower the animation.

Argumentos Opcionais:

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

turtle.**tracer**(*n=None*, *delay=None*)

Parâmetros

- **n** – inteiro não-negativo
- **delay** – inteiro não-negativo

Turn turtle animation on/off and set delay for update drawings. If *n* is given, only each n-th regular screen update is really performed. (Can be used to accelerate the drawing of complex graphics.) When called without arguments, returns the currently stored value of n. Second argument sets delay value (see *delay()*).

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

turtle.**update**()
Perform a TurtleScreen update. To be used when tracer is turned off.

Veja também o método RawTurtle/Turtle *speed()*.

Usando os Eventos de Tela

turtle.**listen**(*xdummy=None*, *ydummy=None*)
Set focus on TurtleScreen (in order to collect key-events). Dummy arguments are provided in order to be able to pass *listen()* to the onclick method.

turtle.**onkey**(*fun*, *key*)
turtle.**onkeyrelease**(*fun*, *key*)

Parâmetros

- **fun** – a function with no arguments or None
- **key** – a string: key (e.g. “a”) or key-symbol (e.g. “space”)

Bind *fun* to key-release event of key. If *fun* is None, event bindings are removed. Remark: in order to be able to register key-events, TurtleScreen must have the focus. (See method *listen()*.)

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onkeypress**(*fun*, *key=None*)

Parâmetros

- **fun** – a function with no arguments or `None`
- **key** – a string: key (e.g. "a") or key-symbol (e.g. "space")

Bind *fun* to key-press event of key if key is given, or to any key-press-event if no key is given. Remark: in order to be able to register key-events, TurtleScreen must have focus. (See method *listen()*.)

```
>>> def f():
...     fd(50)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onclick**(*fun*, *btn=1*, *add=None*)
turtle.**onscreenclick**(*fun*, *btn=1*, *add=None*)

Parâmetros

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – `True` or `False` – if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-click events on this screen. If *fun* is `None`, existing bindings are removed.

Example for a TurtleScreen instance named `screen` and a Turtle instance named turtle:

```
>>> screen.onclick(turtle.goto) # Subsequently clicking into the TurtleScreen will
>>>                             # make the turtle move to the clicked point.
>>> screen.onclick(None)        # remove event binding again
```

Nota: This TurtleScreen method is available as a global function only under the name `onscreenclick`. The global function `onclick` is another one derived from the Turtle method `onclick`.

turtle.**ontimer**(*fun*, *t=0*)

Parâmetros

- **fun** – um função sem nenhum argumento
- **t** – um número >= 0

Install a timer that calls *fun* after *t* milliseconds.

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()   ### makes the turtle march around
>>> running = False
```

turtle.**mainloop**()
turtle.**done**()

Starts event loop - calling Tkinter's mainloop function. Must be the last statement in a turtle graphics program. Must *not* be used if a script is run from within IDLE in -n mode (No subprocess) - for interactive use of turtle graphics.

```
>>> screen.mainloop()
```

Métodos de Entrada

turtle.**textinput**(*title*, *prompt*)

Parâmetros

- **title** – string
- **prompt** – string

Pop up a dialog window for input of a string. Parameter title is the title of the dialog window, prompt is a text mostly describing what information to input. Return the string input. If the dialog is canceled, return `None`.

```
>>> screen.textinput("NIM", "Name of first player:")
```

turtle.**numinput**(*title*, *prompt*, *default=None*, *minval=None*, *maxval=None*)

Parâmetros

- **title** – string
- **prompt** – string
- **default** – número (optional)
- **minval** – número (optional)
- **maxval** – número (optional)

Pop up a dialog window for input of a number. title is the title of the dialog window, prompt is a text mostly describing what numerical information to input. default: default value, minval: minimum value for input, maxval: maximum value for input The number input must be in the range minval .. maxval if these are given. If not, a hint is issued and the dialog remains open for correction. Return the number input. If the dialog is canceled, return `None`.

```
>>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)
```

Configurações e Métodos Especiais

turtle.**mode**(*mode=None*)

Parâmetros **mode** – one of the strings “standard”, “logo” or “world”

Set turtle mode (“standard”, “logo” or “world”) and perform reset. If mode is not given, current mode is returned.

Mode “standard” is compatible with old `turtle`. Mode “logo” is compatible with most Logo turtle graphics. Mode “world” uses user-defined “world coordinates”. **Attention**: in this mode angles appear distorted if `x/y` unit-ratio doesn't equal 1.

Modo	Título inicial da tartaruga	ângulos positivos
“standard”	para a direita (east)	counterclockwise
“logo”	upward (north)	clockwise

```
>>> mode("logo")   # resets turtle heading to north
>>> mode()
'logo'
```

turtle.**colormode**(*cmode=None*)

Parâmetros **cmode** – um dos valroes 1.0 ou 255

Return the colormode or set it to 1.0 or 255. Subsequently *r*, *g*, *b* values of color triples have to be in the range 0..*cmode*.

```
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
     ...
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

turtle.**getcanvas**()

Return the Canvas of this TurtleScreen. Useful for insiders who know what to do with a Tkinter Canvas.

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object ...>
```

turtle.**getshapes**()

Retorna uma lista dos nomes de todas as formas de tartarugas disponíveis no momento.

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

turtle.**register_shape**(*name*, *shape=None*)
turtle.**addshape**(*name*, *shape=None*)

Há três maneiras diferentes de chamar essa função:

(1) *name* is the name of a gif-file and *shape* is `None`: Install the corresponding image shape.

```
>>> screen.register_shape("turtle.gif")
```

Nota: Image shapes *do not* rotate when turning the turtle, so they do not display the heading of the turtle!

(2) *name* is an arbitrary string and *shape* is a tuple of pairs of coordinates: Install the corresponding polygon shape.

```
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))
```

(3) *name* is an arbitrary string and shape is a (compound) `Shape` object: Install the corresponding compound shape.

Add a turtle shape to TurtleScreen's shapelist. Only thusly registered shapes can be used by issuing the command `shape(shapename)`.

`turtle.`**`turtles`**`()`

Retorne uma lista de tartarugas na tela.

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

`turtle.`**`window_height`**`()`

Retorna a altura da janela da tartaruga.

```
>>> screen.window_height()
480
```

`turtle.`**`window_width`**`()`

Retorna a largura da janela da tartaruga.

```
>>> screen.window_width()
640
```

Methods specific to Screen, not inherited from TurtleScreen

`turtle.`**`bye`**`()`

Shut the turtlegraphics window.

`turtle.`**`exitonclick`**`()`

Bind bye() method to mouse clicks on the Screen.

If the value "using_IDLE" in the configuration dictionary is `False` (default value), also enter mainloop. Remark: If IDLE with the `-n` switch (no subprocess) is used, this value should be set to `True` in `turtle.cfg`. In this case IDLE's own mainloop is active also for the client script.

`turtle.`**`setup`**`(`*width=_CFG["width"], height=_CFG["height"], startx=_CFG["leftright"], starty=_CFG["topbottom"]*`)`

Set the size and position of the main window. Default values of arguments are stored in the configuration dictionary and can be changed via a `turtle.cfg` file.

Parâmetros

- **`width`** – if an integer, a size in pixels, if a float, a fraction of the screen; default is 50% of screen

- **height** – if an integer, the height in pixels, if a float, a fraction of the screen; default is 75% of screen
- **startx** – if positive, starting position in pixels from the left edge of the screen, if negative from the right edge, if `None`, center window horizontally
- **starty** – if positive, starting position in pixels from the top edge of the screen, if negative from the bottom edge, if `None`, center window vertically

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>              # sets window to 200x200 pixels, in upper left of screen
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>              # sets window to 75% of screen by 50% of screen and centers
```

`turtle.`**`title`**(*titlestring*)

Parâmetros **titlestring** – a string that is shown in the titlebar of the turtle graphics window

Set title of turtle window to *titlestring*.

```
>>> screen.title("Welcome to the turtle zoo!")
```

25.1.5 Classes Públicas

class `turtle.`**`RawTurtle`**(*canvas*)
class `turtle.`**`RawPen`**(*canvas*)

Parâmetros **canvas** – a `tkinter.Canvas`, a *`ScrolledCanvas`* or a *`TurtleScreen`*

Create a turtle. The turtle has all methods described above as "methods of Turtle/RawTurtle".

class `turtle.`**`Turtle`**
Subclass of RawTurtle, has the same interface but draws on a default *`Screen`* object created automatically when needed for the first time.

class `turtle.`**`TurtleScreen`**(*cv*)

Parâmetros **cv** – uma `tkinter.Canvas`

Provides screen oriented methods like `setbg()` etc. that are described above.

class `turtle.`**`Screen`**
Subclass of TurtleScreen, with *four methods added*.

class `turtle.`**`ScrolledCanvas`**(*master*)

Parâmetros **master** – some Tkinter widget to contain the ScrolledCanvas, i.e. a Tkinter-canvas with scrollbars added

Used by class Screen, which thus automatically provides a ScrolledCanvas as playground for the turtles.

class `turtle.`**`Shape`**(*type_*, *data*)

Parâmetros **type_** – one of the strings "polygon", "image", "compound"

Data structure modeling shapes. The pair `(type_, data)` must follow this specification:

*type_*	*data*
"polygon"	a polygon-tuple, i.e. a tuple of pairs of coordinates
"image"	an image (in this form only used internally!)
"compound"	`None` (a compound shape has to be constructed using the *`addcomponent()`* method)

addcomponent(*poly*, *fill*, *outline=None*)

Parâmetros

- **poly** – a polygon, i.e. a tuple of pairs of numbers
- **fill** – a color the *poly* will be filled with
- **outline** – a color for the poly's outline (if given)

Exemplo:

```
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use register_shape()
```

See *Formas compostas*.

class turtle.**Vec2D**(*x*, *y*)

A two-dimensional vector class, used as a helper class for implementing turtle graphics. May be useful for turtle graphics programs too. Derived from tuple, so a vector is a tuple!

Provides (for *a*, *b* vectors, *k* number):

- `a + b` vetor adicional
- `a - b` subtração de vetor
- `a * b` produto interno
- `k * a` e `a * k` multiplicação com escalar
- `abs(a)` valor absoluto de um
- rotação `a.rotate(angle)`

25.1.6 Ajuda e Configuração

Como usar a Ajuda

The public methods of the Screen and Turtle classes are documented extensively via docstrings. So these can be used as online-help via the Python help facilities:

- When using IDLE, tooltips show the signatures and first lines of the docstrings of typed in function-/method calls.
- Calling *help()* on methods or functions displays the docstrings:

```
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.


      >>> screen.bgcolor("orange")
      >>> screen.bgcolor()
      "orange"
```

(continua na próxima página)

(continuação da página anterior)

```
      >>> screen.bgcolor(0.5,0,0.5)
      >>> screen.bgcolor()
      "#800080"

>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    >>> turtle.penup()
```

- The docstrings of the functions which are derived from methods have a modified form:

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.

    Example::

      >>> bgcolor("orange")
      >>> bgcolor()
      "orange"
      >>> bgcolor(0.5,0,0.5)
      >>> bgcolor()
      "#800080"

>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()
```

These modified docstrings are created automatically together with the function definitions that are derived from the methods at import time.

Translation of docstrings into different languages

There is a utility to create a dictionary the keys of which are the method names and the values of which are the docstrings of the public methods of the classes Screen and Turtle.

`turtle.`**`write_docstringdict`**(*filename="turtle_docstringdict"*)

Parâmetros **`filename`** – a string, used as filename

Create and write docstring-dictionary to a Python script with the given filename. This function has to be called explicitly (it is not used by the turtle graphics classes). The docstring dictionary will be written to the Python script *`filename`*`.py`. It is intended to serve as a template for translation of the docstrings into different languages.

If you (or your students) want to use *`turtle`* with online help in your native language, you have to translate the docstrings and save the resulting file as e.g. `turtle_docstringdict_german.py`.

If you have an appropriate entry in your `turtle.cfg` file this dictionary will be read in at import time and will replace the original English docstrings.

At the time of this writing there are docstring dictionaries in German and in Italian. (Requests please to glingl@aon.at.)

How to configure Screen and Turtles

The built-in default configuration mimics the appearance and behaviour of the old turtle module in order to retain best possible compatibility with it.

If you want to use a different configuration which better reflects the features of this module or which better fits to your needs, e.g. for use in a classroom, you can prepare a configuration file `turtle.cfg` which will be read at import time and modify the configuration according to its settings.

The built in configuration would correspond to the following turtle.cfg:

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

Breve explicação das entradas selecionadas:

- The first four lines correspond to the arguments of the `Screen.setup()` method.
- Line 5 and 6 correspond to the arguments of the method `Screen.screensize()`.
- *shape* can be any of the built-in shapes, e.g: arrow, turtle, etc. For more info try `help(shape)`.

- If you want to use no fillcolor (i.e. make the turtle transparent), you have to write `fillcolor = ""` (but all nonempty strings must not have quotes in the cfg-file).
- If you want to reflect the turtle its state, you have to use `resizemode = auto`.
- If you set e.g. `language = italian` the docstringdict `turtle_docstringdict_italian.py` will be loaded at import time (if present on the import path, e.g. in the same directory as *turtle*.
- The entries *exampleturtle* and *examplescreen* define the names of these objects as they occur in the docstrings. The transformation of method-docstrings to function-docstrings will delete these names from the docstrings.
- *using_IDLE*: Set this to `True` if you regularly work with IDLE and its -n switch ("no subprocess"). This will prevent *exitonclick()* to enter the mainloop.

There can be a `turtle.cfg` file in the directory where *turtle* is stored and an additional one in the current working directory. The latter will override the settings of the first one.

The `Lib/turtledemo` directory contains a `turtle.cfg` file. You can study it as an example and see its effects when running the demos (preferably not from within the demo-viewer).

25.1.7 `turtledemo` — Scripts de Demonstração

The *turtledemo* package includes a set of demo scripts. These scripts can be run and viewed using the supplied demo viewer as follows:

```
python -m turtledemo
```

Alternatively, you can run the demo scripts individually. For example,

```
python -m turtledemo.bytedesign
```

The *turtledemo* package directory contains:

- A demo viewer `__main__.py` which can be used to view the sourcecode of the scripts and run them at the same time.
- Multiple scripts demonstrating different features of the *turtle* module. Examples can be accessed via the Examples menu. They can also be run standalone.
- A `turtle.cfg` file which serves as an example of how to write and use such files.

Os scripts de demonstração são:

Nome	Description (descrição)	Recursos
bytedesign	Padrão de gráficos de tartaruga clássico complexo	`tracer()`, demora, `update()`
chaos	graphs Verhulst dynamics, shows that computer's computations can generate results sometimes against the common sense expectations	coordenadas mundiais
relógio	Relógio analógico que mostra o horário do seu computador	tartarugas como as mãos do relógio, ontimer
colormixer	experimento com r, g, b	`ondrag()`
forest	3 breadth-first trees	randomization
fractalcurves	Curvas de Hilbert & Koch	recursão
lindenmayer	ethnomathematics (indian kolams)	L-System
minimal_hanoi	Torres de Hanoi	Tartarugas retângulos como discos de Hanói (shape, shapesize)
nim	play the classical nim game with three heaps of sticks against the computer.	turtles as nimsticks, event driven (mouse, keyboard)
paint	programa de desenho super minimalista	`onclick()`
peça	elementar	tartaruga: aparência e animação
penrose	aperiodic tiling with kites and darts	`stamp()`
planet_and_moon	simulação do sistema gravitacional	formas compostas, `Vec2D`
round_dance	dancing turtles rotating pairwise in opposite direction	compound shapes, clone shapesize, tilt, get_shapepoly, update
sorting_animate	visual demonstration of different sorting methods	simple alignment, randomization
tree	a (graphical) breadth first tree (using generators)	`clone()`
two_canvases	desenho simples	tartarugas em duas telas
wikipedia	um padrão do artigo wikipedia sobre gráficos de tartaruga	`clone()`, `undo()`
yinyang	outro exemplo elementar	`circle()`

Diverta-se!

25.1.8 Modificações desde a versão do Python 2.6

- The methods `Turtle.tracer()`, `Turtle.window_width()` and `Turtle.window_height()` have been eliminated. Methods with these names and functionality are now available only as methods of `Screen`. The functions derived from these remain available. (In fact already in Python 2.6 these methods were merely duplications of the corresponding `TurtleScreen`/`Screen`-methods.)
- The method `Turtle.fill()` has been eliminated. The behaviour of `begin_fill()` and `end_fill()` have changed slightly: now every filling-process must be completed with an `end_fill()` call.
- A method `Turtle.filling()` has been added. It returns a boolean value: `True` if a filling process is under way, `False` otherwise. This behaviour corresponds to a `fill()` call without arguments in Python 2.6.

25.1.9 Modificações desde a versão do Python 3.0

- The methods `Turtle.shearfactor()`, `Turtle.shapetransform()` and `Turtle.get_shapepoly()` have been added. Thus the full range of regular linear transforms is now available for transforming turtle shapes. `Turtle.tiltangle()` has been enhanced in functionality: it now can be used to get or set the tiltangle. `Turtle.settiltangle()` has been deprecated.
- The method `Screen.onkeypress()` has been added as a complement to `Screen.onkey()` which in fact binds actions to the keyrelease event. Accordingly the latter has got an alias: `Screen.onkeyrelease()`.
- The method `Screen.mainloop()` has been added. So when working only with Screen and Turtle objects one must not additionally import `mainloop()` anymore.
- Two input methods has been added `Screen.textinput()` and `Screen.numinput()`. These popup input dialogs and return strings and numbers respectively.
- Two example scripts `tdemo_nim.py` and `tdemo_round_dance.py` have been added to the `Lib/turtledemo` directory.

25.2 cmd — Suporte para interpretadores de comando orientado a linhas

Código Fonte: Lib/cmd.py

The *Cmd* class provides a simple framework for writing line-oriented command interpreters. These are often useful for test harnesses, administrative tools, and prototypes that will later be wrapped in a more sophisticated interface.

class cmd.**Cmd**(*completekey='tab'*, *stdin=None*, *stdout=None*)

A *Cmd* instance or subclass instance is a line-oriented interpreter framework. There is no good reason to instantiate *Cmd* itself; rather, it's useful as a superclass of an interpreter class you define yourself in order to inherit *Cmd*'s methods and encapsulate action methods.

The optional argument *completekey* is the *readline* name of a completion key; it defaults to `Tab`. If *completekey* is not *None* and *readline* is available, command completion is done automatically.

The optional arguments *stdin* and *stdout* specify the input and output file objects that the Cmd instance or subclass instance will use for input and output. If not specified, they will default to *sys.stdin* and *sys.stdout*.

If you want a given *stdin* to be used, make sure to set the instance's *use_rawinput* attribute to `False`, otherwise *stdin* will be ignored.

25.2.1 Objetos Cmd

A *Cmd* instance has the following methods:

Cmd.**cmdloop**(*intro=None*)

Repeatedly issue a prompt, accept input, parse an initial prefix off the received input, and dispatch to action methods, passing them the remainder of the line as argument.

The optional argument is a banner or intro string to be issued before the first prompt (this overrides the *intro* class attribute).

If the *readline* module is loaded, input will automatically inherit **bash**-like history-list editing (e.g. `Control-P` scrolls back to the last command, `Control-N` forward to the next one, `Control-F` moves the cursor to the right non-destructively, `Control-B` moves the cursor to the left non-destructively, etc.).

An end-of-file on input is passed back as the string `'EOF'`.

An interpreter instance will recognize a command name `foo` if and only if it has a method `do_foo()`. As a special case, a line beginning with the character `'?'` is dispatched to the method `do_help()`. As another special case, a line beginning with the character `'!'` is dispatched to the method `do_shell()` (if such a method is defined).

This method will return when the *postcmd()* method returns a true value. The *stop* argument to *postcmd()* is the return value from the command's corresponding `do_*()` method.

If completion is enabled, completing commands will be done automatically, and completing of commands args is done by calling `complete_foo()` with arguments *text*, *line*, *begidx*, and *endidx*. *text* is the string prefix we are attempting to match: all returned matches must begin with it. *line* is the current input line with leading whitespace removed, *begidx* and *endidx* are the beginning and ending indexes of the prefix text, which could be used to provide different completion depending upon which position the argument is in.

All subclasses of *Cmd* inherit a predefined `do_help()`. This method, called with an argument `'bar'`, invokes the corresponding method `help_bar()`, and if that is not present, prints the docstring of `do_bar()`, if available. With no argument, `do_help()` lists all available help topics (that is, all commands with corresponding `help_*()` methods or commands that have docstrings), and also lists any undocumented commands.

Cmd.**onecmd**(*str*)
Interpret the argument as though it had been typed in response to the prompt. This may be overridden, but should not normally need to be; see the *precmd()* and *postcmd()* methods for useful execution hooks. The return value is a flag indicating whether interpretation of commands by the interpreter should stop. If there is a `do_*()` method for the command *str*, the return value of that method is returned, otherwise the return value from the *default()* method is returned.

Cmd.**emptyline**()
Method called when an empty line is entered in response to the prompt. If this method is not overridden, it repeats the last nonempty command entered.

Cmd.**default**(*line*)
Method called on an input line when the command prefix is not recognized. If this method is not overridden, it prints an error message and returns.

Cmd.**completedefault**(*text*, *line*, *begidx*, *endidx*)
Method called to complete an input line when no command-specific `complete_*()` method is available. By default, it returns an empty list.

Cmd.**precmd**(*line*)
Hook method executed just before the command line *line* is interpreted, but after the input prompt is generated and issued. This method is a stub in *Cmd*; it exists to be overridden by subclasses. The return value is used as the command which will be executed by the *onecmd()* method; the *precmd()* implementation may re-write the command or simply return *line* unchanged.

Cmd.**postcmd**(*stop*, *line*)
Hook method executed just after a command dispatch is finished. This method is a stub in *Cmd*; it exists to be overridden by subclasses. *line* is the command line which was executed, and *stop* is a flag which indicates whether execution will be terminated after the call to *postcmd()*; this will be the return value of the *onecmd()* method. The return value of this method will be used as the new value for the internal flag which corresponds to *stop*; returning false will cause interpretation to continue.

Cmd.**preloop**()
Hook method executed once when *cmdloop()* is called. This method is a stub in *Cmd*; it exists to be overridden by subclasses.

Cmd.**postloop**()
Hook method executed once when *cmdloop()* is about to return. This method is a stub in *Cmd*; it exists to be overridden by subclasses.

Instances of *Cmd* subclasses have some public instance variables:

Cmd.**prompt**
The prompt issued to solicit input.

Cmd.**identchars**
The string of characters accepted for the command prefix.

Cmd.**lastcmd**
The last nonempty command prefix seen.

Cmd.**cmdqueue**
A list of queued input lines. The cmdqueue list is checked in *cmdloop()* when new input is needed; if it is nonempty, its elements will be processed in order, as if entered at the prompt.

Cmd.**intro**
A string to issue as an intro or banner. May be overridden by giving the *cmdloop()* method an argument.

Cmd.**doc_header**
The header to issue if the help output has a section for documented commands.

Cmd.**misc_header**
The header to issue if the help output has a section for miscellaneous help topics (that is, there are `help_*()` methods without corresponding `do_*()` methods).

Cmd.**undoc_header**
The header to issue if the help output has a section for undocumented commands (that is, there are `do_*()` methods without corresponding `help_*()` methods).

Cmd.**ruler**
The character used to draw separator lines under the help-message headers. If empty, no ruler line is drawn. It defaults to `'='`.

Cmd.**use_rawinput**
A flag, defaulting to true. If true, *cmdloop()* uses *input()* to display a prompt and read the next command; if false, `sys.stdout.write()` and `sys.stdin.readline()` are used. (This means that by importing *readline*, on systems that support it, the interpreter will automatically support **Emacs**-like line editing and command-history keystrokes.)

25.2.2 Exemplo do Cmd

The *cmd* module is mainly useful for building custom shells that let a user work with a program interactively.

This section presents a simple example of how to build a shell around a few of the commands in the *turtle* module.

Basic turtle commands such as *forward()* are added to a *Cmd* subclass with method named `do_forward()`. The argument is converted to a number and dispatched to the turtle module. The docstring is used in the help utility provided by the shell.

The example also includes a basic record and playback facility implemented with the *precmd()* method which is responsible for converting the input to lowercase and writing the commands to a file. The `do_playback()` method reads the file and adds the recorded commands to the `cmdqueue` for immediate playback:

```
import cmd, sys
from turtle import *

class TurtleShell(cmd.Cmd):
    intro = 'Welcome to the turtle shell.   Type help or ? to list commands.\n'
    prompt = '(turtle) '
```

(continua na próxima página)

(continuação da página anterior)

```
    file = None

    # ----- basic turtle commands -----
    def do_forward(self, arg):
        'Move the turtle forward by the specified distance:  FORWARD 10'
        forward(*parse(arg))
    def do_right(self, arg):
        'Turn turtle right by given number of degrees:  RIGHT 20'
        right(*parse(arg))
    def do_left(self, arg):
        'Turn turtle left by given number of degrees:  LEFT 90'
        left(*parse(arg))
    def do_goto(self, arg):
        'Move turtle to an absolute position with changing orientation.  GOTO 100 200'
        goto(*parse(arg))
    def do_home(self, arg):
        'Return turtle to the home position:  HOME'
        home()
    def do_circle(self, arg):
        'Draw circle with given radius an options extent and steps:  CIRCLE 50'
        circle(*parse(arg))
    def do_position(self, arg):
        'Print the current turtle position:  POSITION'
        print('Current position is %d %d\n' % position())
    def do_heading(self, arg):
        'Print the current turtle heading in degrees:  HEADING'
        print('Current heading is %d\n' % (heading(),))
    def do_color(self, arg):
        'Set the color:  COLOR BLUE'
        color(arg.lower())
    def do_undo(self, arg):
        'Undo (repeatedly) the last turtle action(s):  UNDO'
    def do_reset(self, arg):
        'Clear the screen and return turtle to center:  RESET'
        reset()
    def do_bye(self, arg):
        'Stop recording, close the turtle window, and exit:  BYE'
        print('Thank you for using Turtle')
        self.close()
        bye()
        return True

    # ----- record and playback -----
    def do_record(self, arg):
        'Save future commands to filename:  RECORD rose.cmd'
        self.file = open(arg, 'w')
    def do_playback(self, arg):
        'Playback commands from a file:  PLAYBACK rose.cmd'
        self.close()
        with open(arg) as f:
            self.cmdqueue.extend(f.read().splitlines())
    def precmd(self, line):
        line = line.lower()
        if self.file and 'playback' not in line:
            print(line, file=self.file)
        return line
    def close(self):
```

(continua na próxima página)

(continuação da página anterior)

```
        if self.file:
            self.file.close()
            self.file = None

def parse(arg):
    'Convert a series of zero or more numbers to an argument tuple'
    return tuple(map(int, arg.split()))

if __name__ == '__main__':
    TurtleShell().cmdloop()
```

Here is a sample session with the turtle shell showing the help functions, using blank lines to repeat commands, and the simple record and playback facility:

```
Welcome to the turtle shell.   Type help or ? to list commands.

(turtle) ?

Documented commands (type help <topic>):
========================================
bye     color    goto     home  playback  record  right
circle  forward  heading  left  position  reset   undo

(turtle) help forward
Move the turtle forward by the specified distance:  FORWARD 10
(turtle) record spiral.cmd
(turtle) position
Current position is 0 0

(turtle) heading
Current heading is 0

(turtle) reset
(turtle) circle 20
(turtle) right 30
(turtle) circle 40
(turtle) right 30
(turtle) circle 60
(turtle) right 30
(turtle) circle 80
(turtle) right 30
(turtle) circle 100
(turtle) right 30
(turtle) circle 120
(turtle) right 30
(turtle) circle 120
(turtle) heading
Current heading is 180

(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 400
```

(continua na próxima página)

(continuação da página anterior)

```
(turtle) right 90
(turtle) forward 500
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 300
(turtle) playback spiral.cmd
Current position is 0 0

Current heading is 0

Current heading is 180

(turtle) bye
Thank you for using Turtle
```

25.3 `shlex` — Análise léxica simples

Código Fonte: Lib/shlex.py

The *shlex* class makes it easy to write lexical analyzers for simple syntaxes resembling that of the Unix shell. This will often be useful for writing minilanguages, (for example, in run control files for Python applications) or for parsing quoted strings.

The *shlex* module defines the following functions:

shlex.**split**(*s*, *comments=False*, *posix=True*)
: Split the string *s* using shell-like syntax. If *comments* is *False* (the default), the parsing of comments in the given string will be disabled (setting the *commenters* attribute of the *shlex* instance to the empty string). This function operates in POSIX mode by default, but uses non-POSIX mode if the *posix* argument is false.

  Nota: Since the *split()* function instantiates a *shlex* instance, passing `None` for *s* will read the string to split from standard input.

  ---

shlex.**quote**(*s*)
: Return a shell-escaped version of the string *s*. The returned value is a string that can safely be used as one token in a shell command line, for cases where you cannot use a list.

  Este isioma seria inseguro:

  ```
  >>> filename = 'somefile; rm -rf ~'
  >>> command = 'ls -l {}'.format(filename)
  >>> print(command)  # executed by a shell: boom!
  ls -l somefile; rm -rf ~
  ```

  quote() lets you plug the security hole:

  ```
  >>> from shlex import quote
  >>> command = 'ls -l {}'.format(quote(filename))
  >>> print(command)
  ls -l 'somefile; rm -rf ~'
  ```

(continua na próxima página)

(continuação da página anterior)

```
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print(remote_command)
ssh home 'ls -l '"'"'somefile; rm -rf ~'"'"''
```

The quoting is compatible with UNIX shells and with *split()*:

```
>>> from shlex import split
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

Novo na versão 3.3.

The *shlex* module defines the following class:

class shlex.**shlex**(*instream=None*, *infile=None*, *posix=False*, *punctuation_chars=False*)

A *shlex* instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with *read()* and *readline()* methods, or a string. If no argument is given, input will be taken from `sys.stdin`. The second optional argument is a filename string, which sets the initial value of the *infile* attribute. If the *instream* argument is omitted or equal to `sys.stdin`, this second argument defaults to "stdin". The *posix* argument defines the operational mode: when *posix* is not true (default), the *shlex* instance will operate in compatibility mode. When operating in POSIX mode, *shlex* will try to be as close as possible to the POSIX shell parsing rules. The *punctuation_chars* argument provides a way to make the behaviour even closer to how real shells parse. This can take a number of values: the default value, `False`, preserves the behaviour seen under Python 3.5 and earlier. If set to `True`, then parsing of the characters `();<>|&` is changed: any run of these characters (considered punctuation characters) is returned as a single token. If set to a non-empty string of characters, those characters will be used as the punctuation characters. Any characters in the *wordchars* attribute that appear in *punctuation_chars* will be removed from *wordchars*. See *Improved Compatibility with Shells* for more information. *punctuation_chars* can be set only upon *shlex* instance creation and can't be modified later.

Alterado na versão 3.6: The *punctuation_chars* parameter was added.

Ver também:

Module *configparser* Parser for configuration files similar to the Windows `.ini` files.

25.3.1 shlex Objects

A *shlex* instance has the following methods:

shlex.**get_token**()

Return a token. If tokens have been stacked using *push_token()*, pop a token off the stack. Otherwise, read one from the input stream. If reading encounters an immediate end-of-file, *eof* is returned (the empty string (`''`) in non-POSIX mode, and `None` in POSIX mode).

shlex.**push_token**(*str*)

Push the argument onto the token stack.

shlex.**read_token**()

Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

shlex.**sourcehook**(*filename*)

When *shlex* detects a source request (see *source* below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (such as `sys.stdin`), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor handles `#include "file.h"`).

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with *open()* called on it to yield the second component. (Note: this is the reverse of the order of arguments in instance initialization!)

This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding 'close' hook, but a shlex instance will call the *close()* method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the *push_source()* and *pop_source()* methods.

shlex.**push_source**(*newstream*, *newfile=None*)

Push an input source stream onto the input stack. If the filename argument is specified it will later be available for use in error messages. This is the same method used internally by the *sourcehook()* method.

shlex.**pop_source**()

Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer reaches EOF on a stacked input stream.

shlex.**error_leader**(*infile=None*, *lineno=None*)

This method generates an error message leader in the format of a Unix C compiler error label; the format is `'"%s", line %d: '`, where the `%s` is replaced with the name of the current source file and the `%d` with the current input line number (the optional arguments can be used to override these).

This convenience is provided to encourage *shlex* users to generate error messages in the standard, parseable format understood by Emacs and other Unix tools.

Instances of *shlex* subclasses have some public instance variables which either control lexical analysis or can be used for debugging:

shlex.**commenters**

The string of characters that are recognized as comment beginners. All characters from the comment beginner to end of line are ignored. Includes just `'#'` by default.

shlex.**wordchars**

The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumerics and underscore. In POSIX mode, the accented characters in the Latin-1 set are also included. If *punctuation_chars* is not empty, the characters `~-./*?=`, which can appear in filename specifications and command line parameters, will also be included in this attribute, and any characters which appear in `punctuation_chars` will be removed from `wordchars` if they are present there.

shlex.**whitespace**

Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

shlex.**escape**

Characters that will be considered as escape. This will be only used in POSIX mode, and includes just `'\'` by default.

shlex.**quotes**

Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By default, includes ASCII single and double quotes.

shlex.**escapedquotes**
Characters in *quotes* that will interpret escape characters defined in *escape*. This is only used in POSIX mode, and includes just `'"'` by default.

shlex.**whitespace_split**
If `True`, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with *shlex*, getting tokens in a similar way to shell arguments. If this attribute is `True`, *punctuation_chars* will have no effect, and splitting will happen only on whitespaces. When using *punctuation_chars*, which is intended to provide parsing closer to that implemented by shells, it is advisable to leave `whitespace_split` as `False` (the default value).

shlex.**infile**
The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

shlex.**instream**
The input stream from which this *shlex* instance is reading characters.

shlex.**source**
This attribute is `None` by default. If you assign a string to it, that string will be recognized as a lexical-level inclusion request similar to the `source` keyword in various shells. That is, the immediately following token will be opened as a filename and input will be taken from that stream until EOF, at which point the *close()* method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

shlex.**debug**
If this attribute is numeric and `1` or more, a *shlex* instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.

shlex.**lineno**
Source line number (count of newlines seen so far plus one).

shlex.**token**
The token buffer. It may be useful to examine this when catching exceptions.

shlex.**eof**
Token used to determine end of file. This will be set to the empty string (`''`), in non-POSIX mode, and to `None` in POSIX mode.

shlex.**punctuation_chars**
A read-only property. Characters that will be considered punctuation. Runs of punctuation characters will be returned as a single token. However, note that no semantic validity checking will be performed: for example, '»>' could be returned as a token, even though it may not be recognised as such by shells.

Novo na versão 3.6.

25.3.2 Regras de análise

When operating in non-POSIX mode, *shlex* will try to obey to the following rules.

- Quote characters are not recognized within words (`Do"Not"Separate` is parsed as the single word `Do"Not"Separate`);
- Escape characters are not recognized;
- Enclosing characters in quotes preserve the literal value of all characters within the quotes;
- Closing quotes separate words (`"Do"Separate` is parsed as `"Do"` and `Separate`);

- If *whitespace_split* is `False`, any character not declared to be a word character, whitespace, or a quote will be returned as a single-character token. If it is `True`, *shlex* will only split words in whitespaces;
- EOF is signaled with an empty string (`''`);
- It's not possible to parse empty strings, even if quoted.

When operating in POSIX mode, *shlex* will try to obey to the following parsing rules.

- Quotes are stripped out, and do not separate words (`"Do"Not"Separate"` is parsed as the single word `DoNotSeparate`);
- Non-quoted escape characters (e.g. `'\'`) preserve the literal value of the next character that follows;
- Enclosing characters in quotes which are not part of *escapedquotes* (e.g. `"'"`) preserve the literal value of all characters within the quotes;
- Enclosing characters in quotes which are part of *escapedquotes* (e.g. `'"'`) preserves the literal value of all characters within the quotes, with the exception of the characters mentioned in *escape*. The escape characters retain its special meaning only when followed by the quote in use, or the escape character itself. Otherwise the escape character will be considered a normal character.
- EOF is signaled with a *None* value;
- Quoted empty strings (`''`) are allowed.

25.3.3 Improved Compatibility with Shells

Novo na versão 3.6.

The *shlex* class provides compatibility with the parsing performed by common Unix shells like `bash`, `dash`, and `sh`. To take advantage of this compatibility, specify the `punctuation_chars` argument in the constructor. This defaults to `False`, which preserves pre-3.6 behaviour. However, if it is set to `True`, then parsing of the characters `();<>|&` is changed: any run of these characters is returned as a single token. While this is short of a full parser for shells (which would be out of scope for the standard library, given the multiplicity of shells out there), it does allow you to perform processing of command lines more easily than you could otherwise. To illustrate, you can see the difference in the following snippet:

```
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"
>>> list(shlex.shlex(text))
['a', '&', '&', 'b', ';', 'c', '&', '&', 'd', '|', '|', 'e', ';', 'f', '>',
"'abc'", ';', '(', 'def', '"ghi"', ')']
>>> list(shlex.shlex(text, punctuation_chars=True))
['a', '&&', 'b', ';', 'c', '&&', 'd', '||', 'e', ';', 'f', '>', "'abc'",
';', '(', 'def', '"ghi"', ')']
```

Of course, tokens will be returned which are not valid for shells, and you'll need to implement your own error checks on the returned tokens.

Instead of passing `True` as the value for the punctuation_chars parameter, you can pass a string with specific characters, which will be used to determine which characters constitute punctuation. For example:

```
>>> import shlex
>>> s = shlex.shlex("a && b || c", punctuation_chars="|")
>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

Nota: When `punctuation_chars` is specified, the *wordchars* attribute is augmented with the characters `~-./*?=`. That is because these characters can appear in file names (including wildcards) and command-line arguments (e.g. `--color=auto`). Hence:

```
>>> import shlex
>>> s = shlex.shlex('~/a && b-c --color=auto || d *.py?',
...                 punctuation_chars=True)
>>> list(s)
['~/a', '&&', 'b-c', '--color=auto', '||', 'd', '*.py?']
```

For best effect, `punctuation_chars` should be set in conjunction with `posix=True`. (Note that `posix=False` is the default for *shlex*.)

CAPÍTULO 26

Interfaces Gráficas de Usuário com Tk

Tk/Tcl tem sido parte integrante do Python. Ele fornece um kit de ferramentas de janela robusto e independente de plataforma, que está disponível para programadores Python usando o pacote *tkinter* e sua extensão, os módulos *tkinter.tix* e *tkinter.ttk*.

O pacote *tkinter* é uma camada fina orientada a objeto sobre Tcl/Tk. Para usar *tkinter*, você não precisa escrever o código Tcl, mas você precisará consultar a documentação do Tk, e ocasionalmente a documentação do Tcl. *tkinter* é um conjunto de wrappers que implementam os widgets Tk como classes Python. Além disso, o módulo interno `_tkinter` fornece um mecanismo de segurança do segmento que permite que o Python e o Tcl interajam.

As principais virtudes do *tkinter* são que ele é rápido e geralmente vem junto com o Python. Embora sua documentação padrão seja fraca, um bom material está disponível, que inclui: referências, tutoriais, um livro e outros. *tkinter* também é famoso por ter uma aparência desatualizada, que foi amplamente melhorada no Tk 8.5. No entanto, existem muitas outras bibliotecas GUI nas quais você poderia se interessar. Para mais informações sobre alternativas, veja a seção *Outros Pacotes de Interface Gráficas de Usuário*.

26.1 : mod: *tkinter* — Interface Python para Tcl / Tk

Código Fonte: Lib/tkinter/__init__.py

O pacote: mod: *tkinter* (" Tk interface ") é a interface padrão do Python para o kit de ferramentas Tk GUI. Ambos Tk e: mod: *tkinter* estão disponíveis na maioria das plataformas Unix, assim como em sistemas Windows. (O próprio Tk não faz parte do Python; ele é mantido no ActiveState.)

Rodar `python -m tkinter` da linha de comando deve abrir uma janela demonstrando uma interface Tk simples, informando que: mod: *tkinter* está apropriadamente instalado em seu sistema, e também mostrando qual versão do Tcl / Tk é instalado, para que você possa ler a documentação do Tcl / Tk específica para essa versão.

Ver também:

Documentação do Tkinter:

Recursos do Python Tkinter <https://wiki.python.org/moin/TkInter> O Python Tkinter Topic Guide fornece uma grande quantidade de informações sobre como usar o Tk do Python e links para outras fontes de informação no Tk.

TKDocs <http://www.tkdocs.com/> Tutorial extenso e páginas de widgets mais amigáveis para alguns dos widgets.

Referência Tkinter 8.5: uma GUI para Python <https://web.archive.org/web/20190524140835/https://infohost.nmt.edu/tcc/help/pubs/t Material de referência on-line.

Tkinter docs from effbot <http://effbot.org/tkinterbook/> Referência online para o tkinter suportado pelo effbot.org.

Programação Python <http://learning-python.com/about-pp4e.html> Livro de Mark Lutz, tem excelente cobertura da Tkinter.

Tkinter moderno para desenvolvedores Python ocupados <https://www.amazon.com/Modern-Tkinter-Python-Developers-ebook/dp/B00 Livro de Mark Roseman sobre a construção de interfaces gráficas de usuário atraentes e modernas com Python e Tkinter.

Programação Python e Tkinter <https://www.manning.com/books/python-and-tkinter-programming> Livro de John Grayson (ISBN 1-884777-81-3).

Documentação Tcl / Tk:

Comandos Tk <https://www.tcl.tk/man/tcl8.6/TkCmd/contents.htm> A maioria dos comandos estão disponíveis como classes: mod: *tkinter* ou: mod:‘ tkinter.ttk‘. Altere ‘8.6’ para corresponder à versão de sua instalação Tcl / Tk.

Páginas de manual recentes do Tcl / Tk <https://www.tcl.tk/doc/> Manuais Tcl / Tk recentes em www.tcl.tk.

Página inicial de ActiveState Tcl O desenvolvimento do Tk/Tcl acontece em larga medida em ActiveState.

Tcl and the Tk Toolkit Livro de John Ousterhout, o inventor do Tcl.

Practical Programming in Tcl and Tk Livro enciclopédico de Brent Welch.

26.1.1 Módulos Tkinter

Na maioria das vezes, `tkinter` é tudo o que você precisa, mas há outros módulos adicionais disponíveis. A interface do Tk está localizada em um módulo binário chamado `_tkinter`. Este módulo contém a interface de baixo nível do Tk, e nunca deve ser utilizada diretamente por aplicativos de programadores. Geralmente é uma biblioteca compartilhada (ou DLL), mas pode estar em alguns casos conectada estaticamente ao interpretador do Python.

Além do módulo de interface do Tk, `tkinter` inclui vários módulos do Python, `tkinter.constants` sendo o mais importante. Importar `tkinter` irá importar `tkinter.constants` automaticamente, então, geralmente, para utilizar o Tkinter tudo o que você ´recisa é de uma simples instrução de importação:

```
import tkinter
```

Ou, com mais frequência:

```
from tkinter import *
```

class `tkinter.`**Tk**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=1*)
A classe `Tk` é instanciada sem argumentos. Isto cria um widget de alto nível de Tk que normalmente é a janela principal da aplicação. Cada instância possui seu próprio interpretador Tcl associado.

`tkinter.`**Tcl**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=0*)
A função `Tcl()` é uma função fábrica que cria um objeto assim como o criado pela classe `Tk`, exceto por não inicializar o subsistema Tk. Isto é útil quando executando o interpretador Tcl em um ambiente onde não se quer criar janelas de alto nível externas, ou quando não se pode (como em sistemas Unix/Linux sem um servidor X).

Um objeto criado pelo objeto *Tcl()* pode ter uma janela de Alto nível criada (e o subssistema Tk inicializado) chamando o método `loadtk()`.

Outros módulos que fornecem suporte Tk incluem:

tkinter.scrolledtext Widget de texto com uma barra de rolagem vertical integrada.

tkinter.colorchooser Caixa de diálogo para permitir que o usuário escolha uma cor.

tkinter.commondialog Classe base para os diálogos definidos nos outros módulos listados aqui.

tkinter.filedialog Diálogos comuns para permitir ao usuário especificar um arquivo para abrir ou salvar.

tkinter.font Utilitários para ajudar a trabalhar com fontes.

tkinter.messagebox Acesso às caixas de diálogo padrão do Tk.

tkinter.simpledialog Diálogos básicos e funções de conveniência.

tkinter.dnd Suporte de arrastar e soltar para: mod: *tkinter*. Isso é experimental e deve se tornar obsoleto quando for substituído pelo Tk DND.

turtle Gráficos de tartaruga em uma janela Tk.

26.1.2 Preservador de vida Tkinter

Esta seção não foi projetada para ser um tutorial completo sobre Tk ou Tkinter. Em vez disso, pretende ser um obstáculo, fornecendo alguma orientação introdutória sobre o sistema.

Créditos:

- Tk foi escrito por John Ousterhout enquanto estava em Berkeley.
- Tkinter foi escrito por Steen Lumholt e Guido van Rossum.
- This Life Preserver foi escrito por Matt Conway, da Universidade de Virgínia.
- A renderização do HTML, com algumas edições livres, foram produzidas de uma versão do FrameMaker por Ken Manheimer.
- Fredrik Lundh elaborou e revisou as descrições de interface de classe, para que elas fiquem atuais com Tk 4.2
- Mike Clarkson converteu a documentação para LaTeX e compilou o capítulo de Interface de Usuário para o manual de referência.

Como usar esta seção

Esta seção foi feita em duas partes: a primeira parte (mais ou menos) cobre os materiais fundamentais, enquanto a segunda parte pode ser levada para o teclado como uma referência útil.

Ao tentar responder perguntas do tipo "como eu faço x", é melhor pensar como eu faço "x" direto no Tk, e então converter de volta para uma chamada correspondente do *tkinter*. Programadores Python normalmente adivinham os comandos corretos em Python apenas olhando a documentação do Tk. Isso significa que para usar Tkinter, você deverá aprender um pouco sobre Tk. Este documento não pode preencher esse papel, então o melhor que podemos fazer é indicar as melhores documentações que existem. Aqui vão algumas dicas:

- Os autores sugerem fortemente que se consiga uma cópia das páginas man do Tk. Especificamente, as páginas man no diretório `manN` são as mais úteis. As páginas man `man3` descrevem a interface C com a biblioteca Tk e, portanto, não são tão úteis para criadores de script.

- Addison-Wesley publica um livro chamado Tcl and the Tk Toolkit de John Ousterhout (ISBN 0-201-63337-X), que é uma boa introdução a Tcl e Tk para o iniciante. O livro não é exaustivo e, para muitos detalhes, ele recorre às páginas de manual.
- `tkinter/__init__.py` é o último recurso para a maioria, mas pode ser um bom lugar para ir quando nada mais faz sentido.

Um simples programa Olá Mundo

```
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack()
        self.create_widgets()

    def create_widgets(self):
        self.hi_there = tk.Button(self)
        self.hi_there["text"] = "Hello World\n(click me)"
        self.hi_there["command"] = self.say_hi
        self.hi_there.pack(side="top")

        self.quit = tk.Button(self, text="QUIT", fg="red",
                              command=self.master.destroy)
        self.quit.pack(side="bottom")

    def say_hi(self):
        print("hi there, everyone!")

root = tk.Tk()
app = Application(master=root)
app.mainloop()
```

26.1.3 Uma olhada (muito) rápida em Tcl / Tk

A hierarquia de classe parece complicada, mas na prática, programadores de aplicação quase sempre se referem às classes nos níveis mais baixos da hierarquia.

Notas:

- Essas classes são fornecidas com o objetivo de organizar certas funções em um espaço de nomes. Elas não devem ser instanciadas de forma independente.
- A classe *Tk* deve ser instanciada apenas uma vez em uma aplicação. Os programadores de aplicação não precisam instanciar uma explicitamente, o sistema cria uma sempre que qualquer uma das outras classes é instanciada.
- A classe `Widget` não foi feita para ser instanciada, é apenas para quando subclasses forem criadas para fazer widgets "reais" (em C++, isto é chamado de *classe abstrata*).

Para fazer uso deste material de referência, haverá vezes em que você precisará saber como ler pequenas passagens de Tk e como identificar as várias partes dos comandos Tk. (Veja a seção *Mapeamento de TK Básico para Tkinter* para equivalentes de *tkinter* do que está mostrado abaixo.)

Scripts Tk são programas Tcl. Como todos os programas Tcl, scripts Tk são apenas listas de elementos sintáticos (tokens) separados por espaços. Um widget Tk é apenas sua *classe*, as *opções* que ajudam a configurá-los, e as *ações* que permitem

fazer coisas úteis.

Para fazer um widget em Tk, o comando é sempre da forma:

```
classCommand newPathname options
```

classCommand denota que tipo de widget fazer (um botão, um rótulo, um menu …)

newPathname É o novo nome para este widget. Todos os nomes em Tk devem ser únicos. Para ajudar a import isto, os widgets em Tk são renomeados com *parêntesis*, assim como arquivos em um sistema de arquivos. O widget de nível mais alto, a *raiz*, é chamado de `.` (ponto) e os filhos são delimitados por demais pontos. Por exemplo, `.myApp.controlPanel.okButton` pode ser o nome de um widget.

options configura a aparência do widget e, em alguns casos, seu comportamento. As opções aparecem na forma de listas de flags ou valores. Flags são precedidas por um '-', como flags em comandos de console em Unix, e os valores são colocados entre aspas se consistirem em mais de uma palavra.

Por exemplo:

```
button   .fred   -fg red -text "hi there"
   ^       ^     \_____________________/
   |       |                |
 class    new            options
command  widget  (-opt val -opt val ...)
```

Uma vez criado, o nome do path do widget se torna um novo comando. Este novo *comando widget* é o manipulador do programador para fazer o novo widget executar alguma *ação*. Em C, você expressaria isso como algumaAção(fred, algumasOpções), em C++, você expressaria isso como fred.algumaAção(algumasOpções), e em Tk, você usaria:

```
.fred someAction someOptions
```

Observe que o nome do objeto, `.fred`, começa com um ponto.

Como você deve esperar, os valores legais de *algumaAção* vão depender da classe do widget: `.fred disable` funciona se fred é um botão (fred fica cinza), mas não funciona se fred é uma etiqueta (desativar etiquetas não é suportado por Tk).

Os valores legais para *algumaOpção* é dependente de ação. Algumas ações, como `disable`, requerem nenhum argumento, outras, como o comando `delete` de uma caixa de entrada de texto, precisaria de argumentos para especificar o intervalo do texto para apagar.

26.1.4 Mapeamento de TK Básico para Tkinter

Comandos de classe em Tk correspondem à construtores de classe em Tkinter.

```
button .fred                =====>  fred = Button()
```

O mestre de um objeto está implícito no novo nome dado a ele no momento da criação. No Tkinter, os masters são especificados explicitamente.

```
button .panel.fred          =====>  fred = Button(panel)
```

As opções de configurações em Tk são dadas como listas de etiquetas com hífen seguidas por valores. Em Tkinter, as opções são especificadas como argumentos de palavra reservada na instância do construtor, e args de palavras reservadas para configurar chamadas, ou como índices de instâncias, na forma de dicionários, para estabelecer instâncias. Veja a seção See section *Opções de configuração* sobre opções de definições.

```
button .fred -fg red         =====>  fred = Button(panel, fg="red")
.fred configure -fg red      =====>  fred["fg"] = red
                             OR ==>  fred.config(fg="red")
```

Em Tk, para executar uma ação em um widget, utilize o nome do widget como comando, seguido pelo nome de uma ação, possivelmente com argumentos (opções). Em Tkinter, você chama métodos na instância da classe para invocar ações no widget. As ações (métodos) que um determindo widget pode executar estão listadas em `tkinter/__init__.py`.

```
.fred invoke                 =====>  fred.invoke()
```

Para dar o widget ao empacotador (gerenciador de geometria), você chama pack com argumentos opcionais. Em Tkinter, a classe Pack detém toda esta funcionalidade, e as várias formas do comando pack estão implementadas como métodos. Todos os widgets em *tkinter* são subclasses de Packer, e portanto herdam todos os métodos de empacotamento. Veja a documentação do módulo *tkinter.tix* para informações adicionais sobre o gerenciador de geometria.

```
pack .fred -side left        =====>  fred.pack(side="left")
```

26.1.5 Como Tk e Tkinter estão relacionados

De cima para baixo:

Sua aplicação aqui (Python) Ums aplicação Python faz uma chamada *tkinter*.

tkinter (pacote Python) Esta chamada (por exemplo, criando um widget de botão), é implementada no pacote *tkinter*, que é escrito em Python. Esta função Python irá analisar os comandos e os argumentos e convertê-los em uma forma que faça parecer que vieram de um script Tk em vez de um script Python.

_tkinter (C) Esses comandos e seus argumentos serão passados para uma função C no módulo de extensão `_tkinter` - note o sublinhado.

Widgets Tk (C e Tcl) Esta função C é capaz de fazer chamadas para outros módulos C, incluindo as funções C que compõem a biblioteca Tk. Tk é implementado em C e algum Tcl. A parte Tcl dos widgets Tk é usada para vincular certos comportamentos padrão aos widgets e é executada uma vez no ponto onde o pacote Python *tkinter* é importado. (O usuário nunca vê este estágio).

Tk (C) A parte Tk dos Widgets Tk implementa o mapeamento final para …

Xlib (C) a biblioteca Xlib para desenhar gráficos na tela.

26.1.6 Referência útil

Opções de configuração

As opções controlam coisas como a cor e a largura da borda de um widget. As opções podem ser definidas de três maneiras:

No momento da criação do objeto, usando argumentos nomeados

```
fred = Button(self, fg="red", bg="blue")
```

Após a criação do objeto, tratando o nome da opção como um índice de dicionário

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

Use o método config() para atualizar vários attrs posteriores à criação do objeto

```
fred.config(fg="red", bg="blue")
```

Para uma explicação completa de uma dada opção e seu comportamento, veja as páginas man do Tk para o widget em questão.

Note que as páginas man listam "OPÇÕES PADRÃO" e "OPÇÕES DE WIDGET ESPECÍFICO" para cada widget. A primeira é uma lista de opções que são comuns a vários widgets, e a segunda são opções que são idiossincráticas ao widget em particular. As Opções Padrão estão documentadas na página man *options(3)*.

Nenhuma distinção entre as opções padrão e específicas de widget são feitas neste documento. Algumas opções não se aplicam a alguns tipos de widgets. A resposta de um dado widget a uma opção particular depende da classe do widget; botões possuem a opção `command`, etiquetas não.

As opções suportadas por um dado widget estão listadas na página man daquele widget, ou podem ser consultadas no tempo de execução chamando o método `config()` sem argumentos, ou chamando o método `keys()` daquele widget. O valor retornado por essas chamadas é um dicionário cujas chaves são os nomes das opções como uma string (por exemplo, `'relief'`) e cujos valores são tuplas de 5 elementos.

Algumas opções, como `bg` são sinônimos para opções comuns com nomes mais longos (`bg` é a abreviação de "background", ou plano de fundo em inglês). Passando o nome de uma opção abreviada ao método `config()` irá retornar uma tupla de 2 elementos, e não uma tupla de 5 elementos. A tupla de 2 elementos devolvida irá conter o nome do sinônimo e a opção "verdadeira" (como por exemplo em `('bg', 'background')`).

Index	Significado	Exemplo
0	nome da opção	`'relief'`
1	nome da opção para buscas de banco de dados	`'relief'`
2	classe de opção para busca de banco de dados	`'Relief'`
3	valor padrão	`'raised'`
4	valor atual	`'groove'`

Exemplo:

```
>>> print(fred.config())
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

É claro que o dicionário exibido irá incluir todas as opções disponíveis e seus valores. Isso foi apenas um exemplo.

O Empacotador

The packer is one of Tk's geometry-management mechanisms. Geometry managers are used to specify the relative positioning of the positioning of widgets within their container - their mutual *master*. In contrast to the more cumbersome *placer* (which is used less commonly, and we do not cover here), the packer takes qualitative relationship specification - *above*, *to the left of*, *filling*, etc - and works everything out to determine the exact placement coordinates for you.

O tamanho de qualquer widget *mestre* é determinado pelo tamanho dos "widgets escravos" internos. O empacotador é usado para controlar onde os widgets escravos aparecem dentro do mestre no qual são empacotados. Você pode empacotar widgets em quadros e quadros em outros quadros, a fim de obter o tipo de layout que deseja. Além disso, o arranjo é ajustado dinamicamente para acomodar alterações incrementais na configuração, uma vez que é empacotado.

Note que os widgets não aparecem até que eles tenham sua geometria especificada pelo gerenciador de geometria. É um erro comum de iniciantes deixar a geometria de fora da especificação, então se surpreender que o widget é criado sendo que nada apareceu. O widget irá aparecer apenas quando tiver, por exemplo, o método método `pack()` do empacotador aplicado a ele.

O método pack() pode ser chamado com pares de palavra reservada-opção/valor que controlam onde o widget deverá aparecer dentro do seu contêiner, e como deverá se comportar quando a janela da aplicação principal for redimensionada. Aqui estão alguns exemplos:

```
fred.pack()                     # defaults to side = "top"
fred.pack(side="left")
fred.pack(expand=1)
```

Opções do Empacotador

Para uma informação mais completa do empacotador e as opções que pode receber, veja as páginas man e a página 183 do livro do John Ousterhout.

âncora Tipo âncora. Denota onde o packer deverá posicionar cada ajudante em seu pacote.

expandir Booleano, `0` ou `1`.

preencher Valores legais: `'x'`, `'y'`, `'both'`, `'none'`.

ipadx e ipady Uma distância - designando o deslocamento interno de cada lado do widget ajudante.

padx and pady Uma distância - designando o deslocamento externo de cada lado do widget ajudante.

side Valores legais são: `'left'`, `'right'`, `'top'`, `'bottom'`.

Acoplando Variáveis de Widgets

A definição do valor atual de alguns widgets (como widgets de entrada de texto) podem ser conectadas diretamente às variáveis da aplicação utilizando métodos especiais. Estas opções são `variable`, `textvariable`, `onvalue`, `offvalue`, e `value`. A conexão funciona por ambos os lados: Se por algum motivo a variável se altera, o widget ao qual ela está conectada irá se atualizar para refletir este novo valor.

Infelizmente, na atual implementação do *`tkinter`* não é possível passar uma variável Python arbitrária para um widget por uma opção `variable` ou `textvariable`. Os únicos tipos de variáveis para as quais isso funciona são variáveis que são subclasses da classe chamada Variable, definida em *`tkinter`*.

Há muitas subclasses de Variable úteis já definidas: `StringVar`, `IntVar`, `DoubleVar`, e `BooleanVar`. Para ler o atual valor de uma variável, chame o método `get()` nelas, e para alterar o valor você deve chamar o método `set()`. Se você seguir este protocolo, o widget irá sempre acompanhar os valores da variável, sem nenhuma intervenção da sua parte.

Por exemplo:

```
class App(Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

        self.entrythingy = Entry()
        self.entrythingy.pack()

        # here is the application variable
        self.contents = StringVar()
        # set it to some value
        self.contents.set("this is a variable")
        # tell the entry widget to watch this variable
        self.entrythingy["textvariable"] = self.contents
```

(continua na próxima página)

(continuação da página anterior)

```
        # and here we get a callback when the user hits return.
        # we will have the program print out the value of the
        # application variable when the user hits return
        self.entrythingy.bind('<Key-Return>',
                              self.print_contents)

    def print_contents(self, event):
        print("hi. contents of entry is now ---->",
              self.contents.get())
```

O Gerenciador de Janela

No Tk, existe um comando utilitário, `wm`, para interagir com o gerenciador de janelas. As opções do comando `wm` permitem que você controle coisas como títulos, localização, ícones bitmap, entre outros. No *tkinter*, estes comandos foram implementados como métodos da classe `Wm` class. Widgets de mais alto nívelsão subclasses da classe `Wm` e portanto podem chamar os métodos `Wm` diretamente.

Para chegar à janela de nível superior que contém um determinado widget, geralmente você pode apenas consultar o mestre do widget. Claro, se o widget foi compactado dentro de um quadro, o mestre não representará uma janela de nível superior. Para chegar à janela de nível superior que contém um widget arbitrário, você pode chamar o método `_root()`. Este método começa com um sublinhado para denotar o fato de que esta função é parte da implementação, e não uma interface para a funcionalidade Tk.

Aqui estão alguns exemplos de uso típico:

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

# create the application
myapp = App()

#
# here are method calls to the window manager class
#
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# start the program
myapp.mainloop()
```

Opções de Tipos de Dados do Tk

âncora Valores legais são os pontos cardeais: `"n"`, `"ne"`, `"e"`, `"se"`, `"s"`, `"sw"`, `"w"`, `"nw"`, e também `"center"`.

bitmap Há 8 bitmaps embutidos nomeados: `'error'`, `'gray25'`, `'gray50'`, `'hourglass'`, `'info'`, `'questhead'`, `'question'`, `'warning'`. Para especificar um nome de arquivo do bitmap X, passe o caminho completo do arquivo, precedido pelo caractere `@`, como em `"@/usr/contrib/bitmap/gumby.bit"`.

booleano Você pode passar os inteiros 0 ou 1 ou as strings `"yes"` ou `"no"`.

callback Esta é uma função Python que não aceita argumentos. Por exemplo:

```
def print_it():
    print("hi there")
fred["command"] = print_it
```

cor As cores podem ser fornecidas como nomes de cores X no arquivo rgb.txt ou como strings que representam valores RGB nos intervalos 4 bits: `"#RGB"`, 8 bits: `"#RRGGBB"`, 12 bits:`"#RRRGGGBBB"` ou 16 bits: `"#RRRRGGGGBBBB"` onde R, G, B aqui representam qualquer dígito hexadecimal legal. Consulte a página 160 do livro de Ousterhout para detalhes.

cursor Os nomes de cursor X padrão de `cursorfont.h` podem ser usados, sem o prefixo `XC_`. Por exemplo, para obter um cursor de mão (`XC_hand2`), use a string `"hand2"`. Você também pode especificar um bitmap e um arquivo de máscara de sua preferência. Veja a página 179 do livro de Ousterhout.

distance As distâncias da tela podem ser especificadas em pixels ou distâncias absolutas. Pixels são dados como números e distâncias absolutas como strings, com os caracteres finais denotando unidades: `c` para centímetros, `i` para polegadas, `m` para milímetros, `p` para os pontos da impressora. Por exemplo, 3,5 polegadas é expresso como `"3.5i"`.

font Tk usa um formato de nome de fonte de lista, como `{courier 10 bold}`. Tamanhos de fonte com números positivos são medidos em pontos; tamanhos com números negativos são medidos em pixels.

geometria Esta é uma string no formato `widthxheight`, onde largura e altura são medidas em pixels para a maioria dos widgets (em caracteres para widgets que exibem texto). Por exemplo: `fred["geometry"] = "200x100"`.

justify Valores aceitos são as strings: `"left"`, `"center"`, `"right"` e `"fill"`.

region Esta é uma string com quatro elementos delimitados por espaço, cada um dos quais a uma distância legal (veja acima). Por exemplo: `"2 3 4 5"` e `"3i 2i 4.5i 2i"` e `"3c 2c 4c 10.43c"` são todas regiões legais.

relief Determina qual será o estilo da borda de um widget. Os valores legais são: `"raised"`, `"sunken"`, `"flat"`, `"groove"` e `"ridge"`.

scrollcommand Este é quase sempre o método `set()` de algum widget da barra de rolagem, mas pode ser qualquer método de widget que receba um único argumento.

wrap: Deve ser um de: `"none"`, `"char"` ou `"word"`.

Ligações e Eventos

O método de ligação do comando widget permite que você observe certos eventos e tenha um acionamento de função de retorno de chamada quando esse tipo de evento ocorrer. A forma do método de ligação é:

```
def bind(self, sequence, func, add=''):
```

sendo que:

sequência é uma string que denota o tipo de evento de destino. (Consulte a página man do bind e a página 201 do livro de John Ousterhout para obter detalhes).

func é uma função Python, tendo um argumento, a ser invocada quando o evento ocorre. Uma instância de evento será passada como argumento. (As funções implantadas dessa forma são comumente conhecidas como *funções de retorno*.)

add é opcional, tanto `''` ou `'+'`. Passar uma string vazia indica que essa ligação substitui todas as outras ligações às quais esse evento está associado. Passar um `'+'` significa que esta função deve ser adicionada à lista de funções ligadas a este tipo de evento.

Por exemplo:

```
def turn_red(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turn_red)
```

Observe como o campo widget do evento está sendo acessado na função de retorno `turn_red()`. Este campo contém o widget que capturou o evento X. A tabela a seguir lista os outros campos de evento que você pode acessar e como eles são denotados no Tk, o que pode ser útil ao se referir às páginas man do Tk.

Tk	Campo de Eventos do Tkinter	Tk	Campo de Eventos do Tkinter
%f	focus	%A	char
%h	height	%E	send_event
%k	keycode	%K	keysym
%s	state	%N	keysym_num
%t	time	%T	tipo
%w	width	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

O Parâmetro index

Vários widgets requerem que parâmetros de "indx" sejam passados. Eles são usados para apontar para um local específico em um widget Text, ou para caracteres específicos em um widget Entry, ou para itens de menu específicos em um widget Menu.

Índices de widget de entrada (índice, índice de visualização, etc.) Os widgets de entrada têm opções que se referem às posições dos caracteres no texto exibido. Você pode acessar esses pontos especiais em um widget de texto usando as seguintes funções *tkinter*:

Índice de widget de texto A notação de índice do widget de texto é muito rica e é melhor detalhada nas páginas do manual Tk.

Índices de menu (menu.invoke(), menu.entryconfig(), etc.) Algumas opções e métodos de menus manipulam entradas de menu específicas. Sempre que um índice de menu é necessário para uma opção ou parâmetro, você pode passar:

- um número inteiro que se refere à posição numérica da entrada no widget, contada do topo, começando com 0;
- a string `"active"`, que se refere à posição do menu que está atualmente sob o cursor;
- a string `"last"` que se refere ao último item do menu;
- Um inteiro precedido por `@`, como em `@6`, onde o inteiro é interpretado como uma coordenada de pixel y no sistema de coordenadas do menu;
- a string `"none"`, que indica nenhuma entrada de menu, mais frequentemente usada com menu.activate() para desativar todas as entradas e, finalmente,
- uma string de texto cujo padrão corresponde ao rótulo da entrada do menu, conforme varrido da parte superior do menu para a parte inferior. Observe que este tipo de índice é considerado após todos os outros, o que significa que as correspondências para itens de menu rotulados como `last`, `active` ou `none` podem ser interpretadas como os literais acima, em vez disso.

Imagens

Imagens de diferentes formatos podem ser criadas por meio da subclasse correspondente de `tkinter.Image`:

- `BitmapImage` para imagens no formato XBM.
- `PhotoImage` para imagens nos formatos PGM, PPM, GIF e PNG. O suporte ao último foi adicionado a partir do Tk 8.6.

Qualquer tipo de imagem é criado através da opção `file` ou `data` (outras opções também estão disponíveis).

O objeto imagem pode então ser usado sempre que uma opção `image` há suporte em algum widget (por exemplo: rótulos, botões, menus). Nestes casos, o Tk não guardará uma referência à imagem. Quando a última referência Python ao objeto imagem for excluída, os dados da imagem também serão excluídos e o Tk exibirá uma caixa vazia onde quer que a imagem tenha sido usada.

Ver também:

O pacote Pillow adiciona suporte para formatos como BMP, JPEG, TIFF e WebP, dentre outros.

26.1.7 Tratadores de arquivos

O Tk permite que você registre e cancele o registro de uma função de retorno que será chamada a partir do laço central do Tk quando uma E/S for possível em um descritor de arquivo. Apenas um tratador pode ser registrado por descritor de arquivo. Código de exemplo:

```
import tkinter
widget = tkinter.Tk()
mask = tkinter.READABLE | tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

Este recurso não está disponível no Windows.

Já que você não sabe quantos bytes estão disponíveis para leitura, você pode não querer usar os métodos *read()* ou *readline()* de *BufferedIOBase* ou *TextIOBase*, já que eles insistirão em ler uma quantidade predefinida de byts. Para sockets, os métodos *recv()* ou *recvfrom()* vão servir; para outros arquivos, use dados brutos ou `os.read(file.fileno(), maxbytecount)`.

Widget.tk.**createfilehandler**(*file*, *mask*, *func*)
Registra a função de retorno do tratador de arquivo *func*. O argumento *file* pode ser um objeto com um método *fileno()* (como um arquivo ou objeto soquete) ou um descritor de arquivo de inteiro. O argumento *mask* é uma combinação OR de qualquer uma das três constantes abaixo. O retorno de chamada é chamado da seguinte maneira:

```
callback(file, mask)
```

Widget.tk.**deletefilehandler**(*file*)
Cancela o registro de um tratador de arquivo.

tkinter.**READABLE**
tkinter.**WRITABLE**
tkinter.**EXCEPTION**
Constantes usadas nos argumentos *mask*.

26.2 : mod: *tkinter.ttk* — Widgets temáticos do Tk

Código Fonte: Lib/tkinter/ttk.py

The *tkinter.ttk* module provides access to the Tk themed widget set, introduced in Tk 8.5. If Python has not been compiled against Tk 8.5, this module can still be accessed if *Tile* has been installed. The former method using Tk 8.5 provides additional benefits including anti-aliased font rendering under X11 and window transparency (requiring a composition window manager on X11).

A ideia básica para: mod: *tkinter.ttk* é separar, na medida do possível, o código que implementa o comportamento de widget do código que implementa sua aparência.

Ver também:

Tk Widget Styling Support Um documento que apresenta o suporte de temas para Tk

26.2.1 Usando Ttk

Para começar a usar Ttk, importe seu modulo:

```
from tkinter import ttk
```

Para substituir os widgets básicos do Tk, a importação deve seguir o Tk import

```
from tkinter import *
from tkinter.ttk import *
```

That code causes several *tkinter.ttk* widgets (`Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale` and `Scrollbar`) to automatically replace the Tk widgets.

This has the direct benefit of using the new widgets which gives a better look and feel across platforms; however, the replacement widgets are not completely compatible. The main difference is that widget options such as "fg", "bg" and others related to widget styling are no longer present in Ttk widgets. Instead, use the `ttk.Style` class for improved styling effects.

Ver também:

Converting existing applications to use Tile widgets A monograph (using Tcl terminology) about differences typically encountered when moving applications to use the new widgets.

26.2.2 Ttk Widgets

Ttk comes with 18 widgets, twelve of which already existed in tkinter: `Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale`, `Scrollbar`, and *`Spinbox`*. The other six are new: *`Combobox`*, *`Notebook`*, *`Progressbar`*, `Separator`, `Sizegrip` and *`Treeview`*. And all them are subclasses of *`Widget`*.

Using the Ttk widgets gives the application an improved look and feel. As discussed above, there are differences in how the styling is coded.

Código Tk:

```
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Código Ttk:

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

For more information about *TtkStyling*, see the *`Style`* class documentation.

26.2.3 Ferramenta

`ttk.Widget` defines standard options and methods supported by Tk themed widgets and is not supposed to be directly instantiated.

Opções padrões

All the `ttk` Widgets accepts the following options:

Opção	Description (descrição)
Classe	Specifies the window class. The class is used when querying the option database for the window's other options, to determine the default bindtags for the window, and to select the widget's default layout and style. This option is read-only, and may only be specified when the window is created.
cursor	Specifies the mouse cursor to be used for the widget. If set to the empty string (the default), the cursor is inherited for the parent widget.
takefocus	Determines whether the window accepts the focus during keyboard traversal. 0, 1 or an empty string is returned. If 0 is returned, it means that the window should be skipped entirely during keyboard traversal. If 1, it means that the window should receive the input focus as long as it is viewable. And an empty string means that the traversal scripts make the decision about whether or not to focus on the window.
estilo	May be used to specify a custom widget style.

Opções de ferramenta rolável

The following options are supported by widgets that are controlled by a scrollbar.

Opção	Description (descrição)
xscrollcommand	Used to communicate with horizontal scrollbars. When the view in the widget's window change, the widget will generate a Tcl command based on the scrollcommand. Usually this option consists of the method `Scrollbar.set()` of some scrollbar. This will cause the scrollbar to be updated whenever the view in the window changes.
yscrollcommand	Used to communicate with vertical scrollbars. For some more information, see above.

Opções de rótulo

The following options are supported by labels, buttons and other button-like widgets.

Opção	Description (descrição)
texto	Specifies a text string to be displayed inside the widget.
textvariable	Specifies a name whose value will be used in place of the text option resource.
underline	If set, specifies the index (0-based) of a character to underline in the text string. The underline character is used for mnemonic activation.
image	Specifies an image to display. This is a list of 1 or more elements. The first element is the default image name. The rest of the list if a sequence of statespec/value pairs as defined by *`Style.map()`*, specifying different images to use when the widget is in a particular state or a combination of states. All images in the list should have the same size.
compound	Specifies how to display the image relative to the text, in the case both text and images options are present. Valid values are: • texto: exibir apenas texto • imagem: exibir apenas a imagem • top, bottom, left, right: display image above, below, left of, or right of the text, respectively. • none: the default. display the image if present, otherwise the text.
width	If greater than zero, specifies how much space, in character widths, to allocate for the text label, if less than zero, specifies a minimum width. If zero or unspecified, the natural width of the text label is used.

Opções de compatibilidade

Opção	Description (descrição)
state	May be set to "normal" or "disabled" to control the "disabled" state bit. This is a write-only option: setting it changes the widget state, but the *`Widget.state()`* method does not affect this option.

Widget States

The widget state is a bitmap of independent state flags.

Flag	Description (descrição)
active	The mouse cursor is over the widget and pressing a mouse button will cause some action to occur
inválido	Widget is disabled under program control
focus	Widget has keyboard focus
pressed	Widget is being pressed
selecionado	"On", "true", or "current" for things like Checkbuttons and radiobuttons
background	Windows and Mac have a notion of an "active" or foreground window. The *background* state is set for widgets in a background window, and cleared for those in the foreground window
readonly	Widget should not allow user modification
alternate	A widget-specific alternate display format
invalid	The widget's value is invalid

A state specification is a sequence of state names, optionally prefixed with an exclamation point indicating that the bit is off.

ttk.Widget

Besides the methods described below, the `ttk.Widget` supports the methods `tkinter.Widget.cget()` and `tkinter.Widget.configure()`.

class tkinter.ttk.**Widget**

identify(*x*, *y*)
Returns the name of the element at position *x* *y*, or the empty string if the point does not lie within any element.

x and *y* are pixel coordinates relative to the widget.

instate(*statespec*, *callback=None*, **args*, ***kw*)
Test the widget's state. If a callback is not specified, returns `True` if the widget state matches *statespec* and `False` otherwise. If callback is specified then it is called with args if widget state matches *statespec*.

state(*statespec=None*)
Modify or inquire widget state. If *statespec* is specified, sets the widget state according to it and return a new *statespec* indicating which flags were changed. If *statespec* is not specified, returns the currently-enabled state flags.

statespec will usually be a list or a tuple.

26.2.4 Combobox

The `ttk.Combobox` widget combines a text field with a pop-down list of values. This widget is a subclass of `Entry`.

Besides the methods inherited from *`Widget`*: `Widget.cget()`, `Widget.configure()`, *`Widget.identify()`*, *`Widget.instate()`* and *`Widget.state()`*, and the following inherited from `Entry`: `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.selection()`, `Entry.xview()`, it has some other methods, described at `ttk.Combobox`.

Opções

This widget accepts the following specific options:

Opção	Description (descrição)
exportselection	Boolean value. If set, the widget selection is linked to the Window Manager selection (which can be returned by invoking Misc.selection_get, for example).
justify	Specifies how the text is aligned within the widget. One of “left”, “center”, or “right”.
height	Specifies the height of the pop-down listbox, in rows.
postcommand	A script (possibly registered with Misc.register) that is called immediately before displaying the values. It may specify which values to display.
state	One of “normal”, “readonly”, or “disabled”. In the “readonly” state, the value may not be edited directly, and the user can only selection of the values from the dropdown list. In the “normal” state, the text field is directly editable. In the “disabled” state, no interaction is possible.
textvariable	Specifies a name whose value is linked to the widget value. Whenever the value associated with that name changes, the widget value is updated, and vice versa. See `tkinter.StringVar`.
values	Specifies the list of values to display in the drop-down listbox.
width	Specifies an integer value indicating the desired width of the entry window, in average-size characters of the widget’s font.

Virtual events

The combobox widgets generates a **«ComboboxSelected»** virtual event when the user selects an element from the list of values.

ttk.Combobox

class tkinter.ttk.**Combobox**

current(*newindex=None*)
If *newindex* is specified, sets the combobox value to the element position *newindex*. Otherwise, returns the index of the current value or -1 if the current value is not in the values list.

get()
Returns the current value of the combobox.

set(*value*)
Sets the value of the combobox to *value*.

26.2.5 Spinbox

The `ttk.Spinbox` widget is a `ttk.Entry` enhanced with increment and decrement arrows. It can be used for numbers or lists of string values. This widget is a subclass of `Entry`.

Besides the methods inherited from *`Widget`*: `Widget.cget()`, `Widget.configure()`, *`Widget.identify()`*, *`Widget.instate()`* and *`Widget.state()`*, and the following inherited from `Entry`: `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.xview()`, it has some other methods, described at `ttk.Spinbox`.

Opções

This widget accepts the following specific options:

Opção	Description (descrição)
de	Float value. If set, this is the minimum value to which the decrement button will decrement. Must be spelled as `from_` when used as an argument, since `from` is a Python keyword.
para	Float value. If set, this is the maximum value to which the increment button will increment.
increment	Float value. Specifies the amount which the increment/decrement buttons change the value. Defaults to 1.0.
values	Sequence of string or float values. If specified, the increment/decrement buttons will cycle through the items in this sequence rather than incrementing or decrementing numbers.
wrap	Boolean value. If `True`, increment and decrement buttons will cycle from the `to` value to the `from` value or the `from` value to the `to` value, respectively.
formato	String value. This specifies the format of numbers set by the increment/decrement buttons. It must be in the form "%W.Pf", where W is the padded width of the value, P is the precision, and '%' and 'f' are literal.
command	Python callable. Will be called with no arguments whenever either of the increment or decrement buttons are pressed.

Virtual events

The spinbox widget generates an **«Increment»** virtual event when the user presses <Up>, and a **«Decrement»** virtual event when the user presses <Down>.

ttk.Spinbox

class tkinter.ttk.**Spinbox**

get()
Returns the current value of the spinbox.

set(*value*)
Sets the value of the spinbox to *value*.

26.2.6 Notebook

Ttk Notebook widget manages a collection of windows and displays a single one at a time. Each child window is associated with a tab, which the user may select to change the currently-displayed window.

Opções

This widget accepts the following specific options:

Opção	Description (descrição)
height	If present and greater than zero, specifies the desired height of the pane area (not including internal padding or tabs). Otherwise, the maximum height of all panes is used.
padding	Specifies the amount of extra space to add around the outside of the notebook. The padding is a list up to four length specifications left top right bottom. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left.
width	If present and greater than zero, specified the desired width of the pane area (not including internal padding). Otherwise, the maximum width of all panes is used.

Tab Options

There are also specific options for tabs:

Opção	Description (descrição)
state	Either “normal”, “disabled” or “hidden”. If “disabled”, then the tab is not selectable. If “hidden”, then the tab is not shown.
sticky	Specifies how the child window is positioned within the pane area. Value is a string containing zero or more of the characters “n”, “s”, “e” or “w”. Each letter refers to a side (north, south, east or west) that the child window will stick to, as per the `grid()` geometry manager.
padding	Specifies the amount of extra space to add between the notebook and this pane. Syntax is the same as for the option padding used by this widget.
texto	Specifies a text to be displayed in the tab.
image	Specifies an image to display in the tab. See the option image described in *`Widget`*.
compound	Specifies how to display the image relative to the text, in the case both options text and image are present. See *Label Options* for legal values.
underline	Specifies the index (0-based) of a character to underline in the text string. The underlined character is used for mnemonic activation if *`Notebook.enable_traversal()`* is called.

Tab Identifiers

The tab_id present in several methods of `ttk.Notebook` may take any of the following forms:

- An integer between zero and the number of tabs
- The name of a child window
- A positional specification of the form “@x,y”, which identifies the tab
- The literal string “current”, which identifies the currently-selected tab
- The literal string “end”, which returns the number of tabs (only valid for *`Notebook.index()`*)

Virtual Events

This widget generates a **«NotebookTabChanged»** virtual event after a new tab is selected.

ttk.Notebook

class tkinter.ttk.**Notebook**

add(*child*, ***kw*)
Adds a new tab to the notebook.

If window is currently managed by the notebook but hidden, it is restored to its previous position.

See *Tab Options* for the list of available options.

forget(*tab_id*)
Removes the tab specified by *tab_id*, unmaps and unmanages the associated window.

hide(*tab_id*)
Hides the tab specified by *tab_id*.

The tab will not be displayed, but the associated window remains managed by the notebook and its configuration remembered. Hidden tabs may be restored with the `add()` command.

identify(*x*, *y*)
Returns the name of the tab element at position *x*, *y*, or the empty string if none.

index(*tab_id*)
Returns the numeric index of the tab specified by *tab_id*, or the total number of tabs if *tab_id* is the string "end".

insert(*pos*, *child*, ***kw*)
Inserts a pane at the specified position.

pos is either the string "end", an integer index, or the name of a managed child. If *child* is already managed by the notebook, moves it to the specified position.

See *Tab Options* for the list of available options.

select(*tab_id=None*)
Selects the specified *tab_id*.

The associated child window will be displayed, and the previously-selected window (if different) is unmapped. If *tab_id* is omitted, returns the widget name of the currently selected pane.

tab(*tab_id*, *option=None*, ***kw*)
Query or modify the options of the specific *tab_id*.

If *kw* is not given, returns a dictionary of the tab option values. If *option* is specified, returns the value of that *option*. Otherwise, sets the options to the corresponding values.

tabs()
Returns a list of windows managed by the notebook.

enable_traversal()
Enable keyboard traversal for a toplevel window containing this notebook.

This will extend the bindings for the toplevel window containing the notebook as follows:

- `Control-Tab`: selects the tab following the currently selected one.
- `Shift-Control-Tab`: selects the tab preceding the currently selected one.

- `Alt-K`: where *K* is the mnemonic (underlined) character of any tab, will select that tab.

Multiple notebooks in a single toplevel may be enabled for traversal, including nested notebooks. However, notebook traversal only works properly if all panes have the notebook they are in as master.

26.2.7 Progressbar

The `ttk.Progressbar` widget shows the status of a long-running operation. It can operate in two modes: 1) the determinate mode which shows the amount completed relative to the total amount of work to be done and 2) the indeterminate mode which provides an animated display to let the user know that work is progressing.

Opções

This widget accepts the following specific options:

Opção	Description (descrição)
orient	One of "horizontal" or "vertical". Specifies the orientation of the progress bar.
length	Specifies the length of the long axis of the progress bar (width if horizontal, height if vertical).
modo	One of "determinate" or "indeterminate".
maximum	A number specifying the maximum value. Defaults to 100.
value	The current value of the progress bar. In "determinate" mode, this represents the amount of work completed. In "indeterminate" mode, it is interpreted as modulo *maximum*; that is, the progress bar completes one "cycle" when its value increases by *maximum*.
variable	A name which is linked to the option value. If specified, the value of the progress bar is automatically set to the value of this name whenever the latter is modified.
phase	Read-only option. The widget periodically increments the value of this option whenever its value is greater than 0 and, in determinate mode, less than maximum. This option may be used by the current theme to provide additional animation effects.

ttk.Progressbar

class tkinter.ttk.**Progressbar**

start(*interval=None*)
Begin autoincrement mode: schedules a recurring timer event that calls *Progressbar.step()* every *interval* milliseconds. If omitted, *interval* defaults to 50 milliseconds.

step(*amount=None*)
Increments the progress bar's value by *amount*.

amount defaults to 1.0 if omitted.

stop()
Stop autoincrement mode: cancels any recurring timer event initiated by *Progressbar.start()* for this progress bar.

26.2.8 Separator

The `ttk.Separator` widget displays a horizontal or vertical separator bar.

It has no other methods besides the ones inherited from `ttk.Widget`.

Opções

This widget accepts the following specific option:

Opção	Description (descrição)
orient	One of "horizontal" or "vertical". Specifies the orientation of the separator.

26.2.9 Sizegrip

The `ttk.Sizegrip` widget (also known as a grow box) allows the user to resize the containing toplevel window by pressing and dragging the grip.

This widget has neither specific options nor specific methods, besides the ones inherited from `ttk.Widget`.

Platform-specific notes

- On MacOS X, toplevel windows automatically include a built-in size grip by default. Adding a `Sizegrip` is harmless, since the built-in grip will just mask the widget.

Bugs

- If the containing toplevel's position was specified relative to the right or bottom of the screen (e.g. ….), the `Sizegrip` widget will not resize the window.
- This widget supports only "southeast" resizing.

26.2.10 Treeview

The `ttk.Treeview` widget displays a hierarchical collection of items. Each item has a textual label, an optional image, and an optional list of data values. The data values are displayed in successive columns after the tree label.

The order in which data values are displayed may be controlled by setting the widget option `displaycolumns`. The tree widget can also display column headings. Columns may be accessed by number or symbolic names listed in the widget option columns. See *Column Identifiers*.

Each item is identified by a unique name. The widget will generate item IDs if they are not supplied by the caller. There is a distinguished root item, named `{}`. The root item itself is not displayed; its children appear at the top level of the hierarchy.

Each item also has a list of tags, which can be used to associate event bindings with individual items and control the appearance of the item.

The Treeview widget supports horizontal and vertical scrolling, according to the options described in *Scrollable Widget Options* and the methods *`Treeview.xview()`* and *`Treeview.yview()`*.

Opções

This widget accepts the following specific options:

Opção	Description (descrição)
columns	A list of column identifiers, specifying the number of columns and their names.
displaycolumns	A list of column identifiers (either symbolic or integer indices) specifying which data columns are displayed and the order in which they appear, or the string "#all".
height	Specifies the number of rows which should be visible. Note: the requested width is determined from the sum of the column widths.
padding	Specifies the internal padding for the widget. The padding is a list of up to four length specifications.
selectmode	Controls how the built-in class bindings manage the selection. One of "extended", "browse" or "none". If set to "extended" (the default), multiple items may be selected. If "browse", only a single item will be selected at a time. If "none", the selection will not be changed. Note that the application code and tag bindings can set the selection however they wish, regardless of the value of this option.
show	A list containing zero or more of the following values, specifying which elements of the tree to display. • tree: display tree labels in column #0. • headings: display the heading row. The default is "tree headings", i.e., show all elements. **Note**: Column #0 always refers to the tree column, even if show="tree" is not specified.

Item Options

The following item options may be specified for items in the insert and item widget commands.

Opção	Description (descrição)
texto	The textual label to display for the item.
image	A Tk Image, displayed to the left of the label.
values	The list of values associated with the item. Each item should have the same number of values as the widget option columns. If there are fewer values than columns, the remaining values are assumed empty. If there are more values than columns, the extra values are ignored.
open	`True`/`False` value indicating whether the item's children should be displayed or hidden.
tags	A list of tags associated with this item.

Tag Options

The following options may be specified on tags:

Opção	Description (descrição)
foreground	Specifies the text foreground color.
background	Specifies the cell or item background color.
font	Specifies the font to use when drawing text.
image	Specifies the item image, in case the item's image option is empty.

Column Identifiers

Column identifiers take any of the following forms:

- A symbolic name from the list of columns option.
- An integer n, specifying the nth data column.
- A string of the form #n, where n is an integer, specifying the nth display column.

Notas:

- Item's option values may be displayed in a different order than the order in which they are stored.
- Column #0 always refers to the tree column, even if show="tree" is not specified.

A data column number is an index into an item's option values list; a display column number is the column number in the tree where the values are displayed. Tree labels are displayed in column #0. If option displaycolumns is not set, then data column n is displayed in column #n+1. Again, **column #0 always refers to the tree column**.

Virtual Events

The Treeview widget generates the following virtual events.

Evento	Description (descrição)
«TreeviewSelect»	Generated whenever the selection changes.
«TreeviewOpen»	Generated just before settings the focus item to open=True.
«TreeviewClose»	Generated just after setting the focus item to open=False.

The `Treeview.focus()` and `Treeview.selection()` methods can be used to determine the affected item or items.

ttk.Treeview

class tkinter.ttk.**Treeview**

bbox(*item*, *column=None*)
Returns the bounding box (relative to the treeview widget's window) of the specified *item* in the form (x, y, width, height).

If *column* is specified, returns the bounding box of that cell. If the *item* is not visible (i.e., if it is a descendant of a closed item or is scrolled offscreen), returns an empty string.

get_children(*item=None*)
Returns the list of children belonging to *item*.

If *item* is not specified, returns root children.

set_children(*item*, **newchildren*)
Replaces *item*'s child with *newchildren*.

Children present in *item* that are not present in *newchildren* are detached from the tree. No items in *newchildren* may be an ancestor of *item*. Note that not specifying *newchildren* results in detaching *item*'s children.

column(*column*, *option=None*, ***kw*)
Query or modify the options for the specified *column*.

If *kw* is not given, returns a dict of the column option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- **id** Returns the column name. This is a read-only option.
- **anchor: One of the standard Tk anchor values.** Specifies how the text in this column should be aligned with respect to the cell.
- **minwidth: width** The minimum width of the column in pixels. The treeview widget will not make the column any smaller than specified by this option when the widget is resized or the user drags a column.
- **stretch: `True/False`** Specifies whether the column's width should be adjusted when the widget is resized.
- **width: width** The width of the column in pixels.

To configure the tree column, call this with column = "#0"

delete(**items*)
Delete all specified *items* and all their descendants.

The root item may not be deleted.

detach(**items*)
Unlinks all of the specified *items* from the tree.

The items and all of their descendants are still present, and may be reinserted at another point in the tree, but will not be displayed.

The root item may not be detached.

exists(*item*)
Returns `True` if the specified *item* is present in the tree.

focus(*item=None*)
If *item* is specified, sets the focus item to *item*. Otherwise, returns the current focus item, or '' if there is none.

heading(*column*, *option=None*, ***kw*)
Query or modify the heading options for the specified *column*.

If *kw* is not given, returns a dict of the heading option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- **text: text** The text to display in the column heading.
- **image: imageName** Specifies an image to display to the right of the column heading.

- **anchor: anchor** Specifies how the heading text should be aligned. One of the standard Tk anchor values.
- **command: callback** A callback to be invoked when the heading label is pressed.

To configure the tree column heading, call this with column = "#0".

identify (*component*, *x*, *y*)
Returns a description of the specified *component* under the point given by *x* and *y*, or the empty string if no such *component* is present at that position.

identify_row (*y*)
Returns the item ID of the item at position *y*.

identify_column (*x*)
Returns the data column identifier of the cell at position *x*.

The tree column has ID #0.

identify_region (*x*, *y*)
Returns one of:

region	meaning
heading	Tree heading area.
separator	Space between two columns headings.
tree	The tree area.
cell	A data cell.

Availability: Tk 8.6.

identify_element (*x*, *y*)
Returns the element at position *x*, *y*.

Availability: Tk 8.6.

index (*item*)
Returns the integer index of *item* within its parent's list of children.

insert (*parent*, *index*, *iid=None*, ***kw*)
Creates a new item and returns the item identifier of the newly created item.

parent is the item ID of the parent item, or the empty string to create a new top-level item. *index* is an integer, or the value "end", specifying where in the list of parent's children to insert the new item. If *index* is less than or equal to zero, the new node is inserted at the beginning; if *index* is greater than or equal to the current number of children, it is inserted at the end. If *iid* is specified, it is used as the item identifier; *iid* must not already exist in the tree. Otherwise, a new unique identifier is generated.

See *Item Options* for the list of available points.

item (*item*, *option=None*, ***kw*)
Query or modify the options for the specified *item*.

If no options are given, a dict with options/values for the item is returned. If *option* is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values as given by *kw*.

move (*item*, *parent*, *index*)
Moves *item* to position *index* in *parent*'s list of children.

It is illegal to move an item under one of its descendants. If *index* is less than or equal to zero, *item* is moved to the beginning; if greater than or equal to the number of children, it is moved to the end. If *item* was detached it is reattached.

next (*item*)
Returns the identifier of *item*'s next sibling, or '' if *item* is the last child of its parent.

parent (*item*)
Returns the ID of the parent of *item*, or '' if *item* is at the top level of the hierarchy.

prev (*item*)
Returns the identifier of *item*'s previous sibling, or '' if *item* is the first child of its parent.

reattach (*item*, *parent*, *index*)
An alias for `Treeview.move()`.

see (*item*)
Ensure that *item* is visible.

Sets all of *item*'s ancestors open option to `True`, and scrolls the widget if necessary so that *item* is within the visible portion of the tree.

selection (*selop=None*, *items=None*)
If *selop* is not specified, returns selected items. Otherwise, it will act according to the following selection methods.

Deprecated since version 3.6, will be removed in version 3.8: Using `selection()` for changing the selection state is deprecated. Use the following selection methods instead.

selection_set (**items*)
items becomes the new selection.

Alterado na versão 3.6: *items* can be passed as separate arguments, not just as a single tuple.

selection_add (**items*)
Add *items* to the selection.

Alterado na versão 3.6: *items* can be passed as separate arguments, not just as a single tuple.

selection_remove (**items*)
Remove *items* from the selection.

Alterado na versão 3.6: *items* can be passed as separate arguments, not just as a single tuple.

selection_toggle (**items*)
Toggle the selection state of each item in *items*.

Alterado na versão 3.6: *items* can be passed as separate arguments, not just as a single tuple.

set (*item*, *column=None*, *value=None*)
With one argument, returns a dictionary of column/value pairs for the specified *item*. With two arguments, returns the current value of the specified *column*. With three arguments, sets the value of given *column* in given *item* to the specified *value*.

tag_bind (*tagname*, *sequence=None*, *callback=None*)
Bind a callback for the given event *sequence* to the tag *tagname*. When an event is delivered to an item, the callbacks for each of the item's tags option are called.

tag_configure (*tagname*, *option=None*, ***kw*)
Query or modify the options for the specified *tagname*.

If *kw* is not given, returns a dict of the option settings for *tagname*. If *option* is specified, returns the value for that *option* for the specified *tagname*. Otherwise, sets the options to the corresponding values for the given *tagname*.

tag_has (*tagname*, *item=None*)
If *item* is specified, returns 1 or 0 depending on whether the specified *item* has the given *tagname*. Otherwise, returns a list of all items that have the specified tag.

Availability: Tk 8.6

xview (**args*)
Query or modify horizontal position of the treeview.

yview (**args*)
Query or modify vertical position of the treeview.

26.2.11 Ttk Styling

Each widget in `ttk` is assigned a style, which specifies the set of elements making up the widget and how they are arranged, along with dynamic and default settings for element options. By default the style name is the same as the widget's class name, but it may be overridden by the widget's style option. If you don't know the class name of a widget, use the method `Misc.winfo_class()` (somewidget.winfo_class()).

Ver também:

Tcl'2004 conference presentation This document explains how the theme engine works

class tkinter.ttk.**Style**
This class is used to manipulate the style database.

configure (*style*, *query_opt=None*, ***kw*)
Query or set the default value of the specified option(s) in *style*.

Each key in *kw* is an option and each value is a string identifying the value for that option.

For example, to change every default button to be a flat button with some padding and a different background color:

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

ttk.Style().configure("TButton", padding=6, relief="flat",
   background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

map (*style*, *query_opt=None*, ***kw*)
Query or sets dynamic values of the specified option(s) in *style*.

Each key in *kw* is an option and each value should be a list or a tuple (usually) containing statespecs grouped in tuples, lists, or some other preference. A statespec is a compound of one or more states and then a value.

An example may make it more understandable:

```
import tkinter
from tkinter import ttk

root = tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
```

(continua na próxima página)

(continuação da página anterior)

```
    background=[('pressed', '!disabled', 'black'), ('active', 'white')]
    )

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()

root.mainloop()
```

Note that the order of the (states, value) sequences for an option does matter, if the order is changed to `[('active', 'blue'), ('pressed', 'red')]` in the foreground option, for example, the result would be a blue foreground when the widget were in active or pressed states.

lookup (*style*, *option*, *state=None*, *default=None*)
Returns the value specified for *option* in *style*.

If *state* is specified, it is expected to be a sequence of one or more states. If the *default* argument is set, it is used as a fallback value in case no specification for option is found.

To check what font a Button uses by default:

```
from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))
```

layout (*style*, *layoutspec=None*)
Define the widget layout for given *style*. If *layoutspec* is omitted, return the layout specification for given style.

layoutspec, if specified, is expected to be a list or some other sequence type (excluding strings), where each item should be a tuple and the first item is the layout name and the second item should have the format described in *Layouts*.

To understand the format, see the following example (it is not intended to do anything useful):

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.layout("TMenubutton", [
   ("Menubutton.background", None),
   ("Menubutton.button", {"children":
       [("Menubutton.focus", {"children":
           [("Menubutton.padding", {"children":
               [("Menubutton.label", {"side": "left", "expand": 1})]
           })]
       })]
   }),
])

mbtn = ttk.Menubutton(text='Text')
mbtn.pack()
root.mainloop()
```

element_create (*elementname*, *etype*, **args*, ***kw*)
Create a new element in the current theme, of the given *etype* which is expected to be either “image”, “from” or “vsapi”. The latter is only available in Tk 8.6a for Windows XP and Vista and is not described here.

If “image” is used, *args* should contain the default image name followed by statespec/value pairs (this is the imagespec), and *kw* may have the following options:

- **border=padding** padding is a list of up to four integers, specifying the left, top, right, and bottom borders, respectively.
- **height=height** Specifies a minimum height for the element. If less than zero, the base image's height is used as a default.
- **padding=padding** Specifies the element's interior padding. Defaults to border's value if not specified.
- **sticky=spec** Specifies how the image is placed within the final parcel. spec contains zero or more characters "n", "s", "w", or "e".
- **width=width** Specifies a minimum width for the element. If less than zero, the base image's width is used as a default.

If "from" is used as the value of *etype*, `element_create()` will clone an existing element. *args* is expected to contain a themename, from which the element will be cloned, and optionally an element to clone from. If this element to clone from is not specified, an empty element will be used. *kw* is discarded.

element_names()
Returns the list of elements defined in the current theme.

element_options(*elementname*)
Returns the list of *elementname*'s options.

theme_create(*themename*, *parent=None*, *settings=None*)
Create a new theme.

It is an error if *themename* already exists. If *parent* is specified, the new theme will inherit styles, elements and layouts from the parent theme. If *settings* are present they are expected to have the same syntax used for `theme_settings()`.

theme_settings(*themename*, *settings*)
Temporarily sets the current theme to *themename*, apply specified *settings* and then restore the previous theme.

Each key in *settings* is a style and each value may contain the keys 'configure', 'map', 'layout' and 'element create' and they are expected to have the same format as specified by the methods `Style.configure()`, `Style.map()`, `Style.layout()` and `Style.element_create()` respectively.

As an example, let's change the Combobox for the default theme a bit:

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
   "TCombobox": {
       "configure": {"padding": 5},
       "map": {
           "background": [("active", "green2"),
                          ("!disabled", "green4")],
           "fieldbackground": [("!disabled", "green3")],
           "foreground": [("focus", "OliveDrab1"),
                          ("!disabled", "OliveDrab2")]
       }
   }
})

combo = ttk.Combobox().pack()
```

(continua na próxima página)

(continuação da página anterior)

```
root.mainloop()
```

theme_names()
Returns a list of all known themes.

theme_use(*themename=None*)
If *themename* is not given, returns the theme in use. Otherwise, sets the current theme to *themename*, refreshes all widgets and emits a «ThemeChanged» event.

Layouts

A layout can be just `None`, if it takes no options, or a dict of options specifying how to arrange the element. The layout mechanism uses a simplified version of the pack geometry manager: given an initial cavity, each element is allocated a parcel. Valid options/values are:

- **side: whichside** Specifies which side of the cavity to place the element; one of top, right, bottom or left. If omitted, the element occupies the entire cavity.
- **sticky: nswe** Specifies where the element is placed inside its allocated parcel.
- **unit: 0 or 1** If set to 1, causes the element and all of its descendants to be treated as a single element for the purposes of *Widget.identify()* et al. It's used for things like scrollbar thumbs with grips.
- **children: [sublayout...]** Specifies a list of elements to place inside the element. Each element is a tuple (or other sequence type) where the first item is the layout name, and the other is a *Layout*.

26.3 tkinter.tix — Extension widgets for Tk

Código Fonte: Lib/tkinter/tix.py

Obsoleto desde a versão 3.6: This Tk extension is unmaintained and should not be used in new code. Use *tkinter.ttk* instead.

The *tkinter.tix* (Tk Interface Extension) module provides an additional rich set of widgets. Although the standard Tk library has many useful widgets, they are far from complete. The *tkinter.tix* library provides most of the commonly needed widgets that are missing from standard Tk: *HList*, *ComboBox*, *Control* (a.k.a. SpinBox) and an assortment of scrollable widgets. *tkinter.tix* also includes many more widgets that are generally useful in a wide range of applications: *NoteBook*, *FileEntry*, *PanedWindow*, etc; there are more than 40 of them.

With all these new widgets, you can introduce new interaction techniques into applications, creating more useful and more intuitive user interfaces. You can design your application by choosing the most appropriate widgets to match the special needs of your application and users.

Ver também:

Tix Homepage The home page for `Tix`. This includes links to additional documentation and downloads.

Tix Man Pages On-line version of the man pages and reference material.

Tix Programming Guide On-line version of the programmer's reference material.

Tix Development Applications Tix applications for development of Tix and Tkinter programs. Tide applications work under Tk or Tkinter, and include **TixInspect**, an inspector to remotely modify and debug Tix/Tk/Tkinter applications.

26.3.1 Using Tix

class tkinter.tix.**Tk**(*screenName=None*, *baseName=None*, *className='Tix'*)
Toplevel widget of Tix which represents mostly the main window of an application. It has an associated Tcl interpreter.

Classes in the *tkinter.tix* module subclasses the classes in the *tkinter*. The former imports the latter, so to use *tkinter.tix* with Tkinter, all you need to do is to import one module. In general, you can just import *tkinter.tix*, and replace the toplevel call to *tkinter.Tk* with `tix.Tk`:

```
from tkinter import tix
from tkinter.constants import *
root = tix.Tk()
```

To use *tkinter.tix*, you must have the Tix widgets installed, usually alongside your installation of the Tk widgets. To test your installation, try the following:

```
from tkinter import tix
root = tix.Tk()
root.tk.eval('package require Tix')
```

26.3.2 Tix Widgets

Tix introduces over 40 widget classes to the *tkinter* repertoire.

Widgets básicos

class tkinter.tix.**Balloon**
A Balloon that pops up over a widget to provide help. When the user moves the cursor inside a widget to which a Balloon widget has been bound, a small pop-up window with a descriptive message will be shown on the screen.

class tkinter.tix.**ButtonBox**
The ButtonBox widget creates a box of buttons, such as is commonly used for `Ok Cancel`.

class tkinter.tix.**ComboBox**
The ComboBox widget is similar to the combo box control in MS Windows. The user can select a choice by either typing in the entry subwidget or selecting from the listbox subwidget.

class tkinter.tix.**Control**
The Control widget is also known as the `SpinBox` widget. The user can adjust the value by pressing the two arrow buttons or by entering the value directly into the entry. The new value will be checked against the user-defined upper and lower limits.

class tkinter.tix.**LabelEntry**
The LabelEntry widget packages an entry widget and a label into one mega widget. It can be used to simplify the creation of "entry-form" type of interface.

class tkinter.tix.**LabelFrame**
The LabelFrame widget packages a frame widget and a label into one mega widget. To create widgets inside a LabelFrame widget, one creates the new widgets relative to the `frame` subwidget and manage them inside the `frame` subwidget.

class tkinter.tix.**Meter**
The Meter widget can be used to show the progress of a background job which may take a long time to execute.

class tkinter.tix.**OptionMenu**
The OptionMenu creates a menu button of options.

class tkinter.tix.**PopupMenu**
The PopupMenu widget can be used as a replacement of the `tk_popup` command. The advantage of the `Tix` *PopupMenu* widget is it requires less application code to manipulate.

class tkinter.tix.**Select**
The Select widget is a container of button subwidgets. It can be used to provide radio-box or check-box style of selection options for the user.

class tkinter.tix.**StdButtonBox**
The StdButtonBox widget is a group of standard buttons for Motif-like dialog boxes.

Seletores de arquivo

class tkinter.tix.**DirList**
The DirList widget displays a list view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class tkinter.tix.**DirTree**
The DirTree widget displays a tree view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class tkinter.tix.**DirSelectDialog**
The DirSelectDialog widget presents the directories in the file system in a dialog window. The user can use this dialog window to navigate through the file system to select the desired directory.

class tkinter.tix.**DirSelectBox**
The *DirSelectBox* is similar to the standard Motif(TM) directory-selection box. It is generally used for the user to choose a directory. DirSelectBox stores the directories mostly recently selected into a ComboBox widget so that they can be quickly selected again.

class tkinter.tix.**ExFileSelectBox**
The ExFileSelectBox widget is usually embedded in a tixExFileSelectDialog widget. It provides a convenient method for the user to select files. The style of the *ExFileSelectBox* widget is very similar to the standard file dialog on MS Windows 3.1.

class tkinter.tix.**FileSelectBox**
The FileSelectBox is similar to the standard Motif(TM) file-selection box. It is generally used for the user to choose a file. FileSelectBox stores the files mostly recently selected into a *ComboBox* widget so that they can be quickly selected again.

class tkinter.tix.**FileEntry**
The FileEntry widget can be used to input a filename. The user can type in the filename manually. Alternatively, the user can press the button widget that sits next to the entry, which will bring up a file selection dialog.

Hierarchical ListBox

class tkinter.tix.**HList**
The HList widget can be used to display any data that have a hierarchical structure, for example, file system directory trees. The list entries are indented and connected by branch lines according to their places in the hierarchy.

class tkinter.tix.**CheckList**
The CheckList widget displays a list of items to be selected by the user. CheckList acts similarly to the Tk check-button or radiobutton widgets, except it is capable of handling many more items than checkbuttons or radiobuttons.

class tkinter.tix.**Tree**
The Tree widget can be used to display hierarchical data in a tree form. The user can adjust the view of the tree by opening or closing parts of the tree.

Tabular ListBox

class tkinter.tix.**TList**
The TList widget can be used to display data in a tabular format. The list entries of a *TList* widget are similar to the entries in the Tk listbox widget. The main differences are (1) the *TList* widget can display the list entries in a two dimensional format and (2) you can use graphical images as well as multiple colors and fonts for the list entries.

Manager Widgets

class tkinter.tix.**PanedWindow**
The PanedWindow widget allows the user to interactively manipulate the sizes of several panes. The panes can be arranged either vertically or horizontally. The user changes the sizes of the panes by dragging the resize handle between two panes.

class tkinter.tix.**ListNoteBook**
The ListNoteBook widget is very similar to the `TixNoteBook` widget: it can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages (windows). At one time only one of these pages can be shown. The user can navigate through these pages by choosing the name of the desired page in the `hlist` subwidget.

class tkinter.tix.**NoteBook**
The NoteBook widget can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages. At one time only one of these pages can be shown. The user can navigate through these pages by choosing the visual "tabs" at the top of the NoteBook widget.

Image Types

The *tkinter.tix* module adds:

- pixmap capabilities to all *tkinter.tix* and *tkinter* widgets to create color images from XPM files.
- Compound image types can be used to create images that consists of multiple horizontal lines; each line is composed of a series of items (texts, bitmaps, images or spaces) arranged from left to right. For example, a compound image can be used to display a bitmap and a text string simultaneously in a Tk `Button` widget.

Miscellaneous Widgets

class tkinter.tix.**InputOnly**
The InputOnly widgets are to accept inputs from the user, which can be done with the `bind` command (Unix only).

Form Geometry Manager

In addition, *tkinter.tix* augments *tkinter* by providing:

class tkinter.tix.**Form**
The Form geometry manager based on attachment rules for all Tk widgets.

26.3.3 Comandos Tix

class tkinter.tix.**tixCommand**

The tix commands provide access to miscellaneous elements of `Tix`'s internal state and the `Tix` application context. Most of the information manipulated by these methods pertains to the application as a whole, or to a screen or display, rather than to a particular window.

To view the current settings, the common usage is:

```
from tkinter import tix
root = tix.Tk()
print(root.tix_configure())
```

tixCommand.**tix_configure**(*cnf=None*, ***kw*)

Query or modify the configuration options of the Tix application context. If no option is specified, returns a dictionary all of the available options. If option is specified with no value, then the method returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the method modifies the given option(s) to have the given value(s); in this case the method returns an empty string. Option may be any of the configuration options.

tixCommand.**tix_cget**(*option*)

Returns the current value of the configuration option given by *option*. Option may be any of the configuration options.

tixCommand.**tix_getbitmap**(*name*)

Locates a bitmap file of the name `name.xpm` or `name` in one of the bitmap directories (see the *tix_addbitmapdir()* method). By using *tix_getbitmap()*, you can avoid hard coding the pathnames of the bitmap files in your application. When successful, it returns the complete pathname of the bitmap file, prefixed with the character `@`. The returned value can be used to configure the `bitmap` option of the Tk and Tix widgets.

tixCommand.**tix_addbitmapdir**(*directory*)

Tix maintains a list of directories under which the *tix_getimage()* and *tix_getbitmap()* methods will search for image files. The standard bitmap directory is `$TIX_LIBRARY/bitmaps`. The *tix_addbitmapdir()* method adds *directory* into this list. By using this method, the image files of an applications can also be located using the *tix_getimage()* or *tix_getbitmap()* method.

tixCommand.**tix_filedialog**([*dlgclass*])

Returns the file selection dialog that may be shared among different calls from this application. This method will create a file selection dialog widget when it is called the first time. This dialog will be returned by all subsequent calls to *tix_filedialog()*. An optional dlgclass parameter can be passed as a string to specified what type of file selection dialog widget is desired. Possible options are `tix`, `FileSelectDialog` or `tixExFileSelectDialog`.

tixCommand.**tix_getimage**(*self*, *name*)

Locates an image file of the name `name.xpm`, `name.xbm` or `name.ppm` in one of the bitmap directories (see the *tix_addbitmapdir()* method above). If more than one file with the same name (but different extensions) exist, then the image type is chosen according to the depth of the X display: xbm images are chosen on monochrome displays and color images are chosen on color displays. By using *tix_getimage()*, you can avoid hard coding the pathnames of the image files in your application. When successful, this method returns the name of the newly created image, which can be used to configure the `image` option of the Tk and Tix widgets.

tixCommand.**tix_option_get**(*name*)

Gets the options maintained by the Tix scheme mechanism.

tixCommand.**tix_resetoptions**(*newScheme*, *newFontSet*[, *newScmPrio*])

Resets the scheme and fontset of the Tix application to *newScheme* and *newFontSet*, respectively. This affects only

those widgets created after this call. Therefore, it is best to call the resetoptions method before the creation of any widgets in a Tix application.

The optional parameter *newScmPrio* can be given to reset the priority level of the Tk options set by the Tix schemes.

Because of the way Tk handles the X option database, after Tix has been has imported and inited, it is not possible to reset the color schemes and font sets using the `tix_config()` method. Instead, the *`tix_resetoptions()`* method must be used.

26.4 `tkinter.scrolledtext` — Wdiget Scrolled Text

Código-fonte: Lib/tkinter/scrolledtext.py

O módulo *`tkinter.scrolledtext`* fornece uma classe com o mesmo nome que implementa um widget de texto básico que possui uma barra de rolagem vertical configurada para fazer a "coisa certa". Usar a classe `ScrolledText` é muito mais fácil do que configurar um widget de texto e barra de rolagem diretamente. O construtor é o mesmo da classe `tkinter.Text`.

O widget de texto e a barra de rolagem são agrupados em `Frame`, e os métodos dos gerenciadores de geometria `Grid` e `Pack` são adquiridos do objeto `Frame`. Isso permite que o widget `ScrolledText` seja usado diretamente para obter o comportamento mais normal de gerenciamento de geometria.

Se um controle mais específico for necessário, os seguintes atributos estarão disponíveis:

`ScrolledText.`**`frame`**
: O quadro que envolve os widgets de barra de rolagem e texto.

`ScrolledText.`**`vbar`**
: O widget da barra de rolagem.

26.5 IDLE

Código Fonte: Lib/idlelib/

IDLE is Python's Integrated Development and Learning Environment.

IDLE has the following features:

- coded in 100% pure Python, using the *`tkinter`* GUI toolkit
- cross-platform: works mostly the same on Windows, Unix, and macOS
- Python shell window (interactive interpreter) with colorizing of code input, output, and error messages
- multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features
- search within any window, replace within editor windows, and search through multiple files (grep)
- debugger with persistent breakpoints, stepping, and viewing of global and local namespaces
- configuration, browsers, and other dialogs

26.5.1 Menus

IDLE has two main window types, the Shell window and the Editor window. It is possible to have multiple editor windows simultaneously. On Windows and Linux, each has its own top menu. Each menu documented below indicates which window type it is associated with.

Output windows, such as used for Edit => Find in Files, are a subtype of editor window. They currently have the same top menu but a different default title and context menu.

On macOS, there is one application menu. It dynamically changes according to the window currently selected. It has an IDLE menu, and some entries described below are moved around to conform to Apple guidelines.

Menu Arquivo (Console e Editor)

Novo Arquivo Create a new file editing window.

Abrir… Open an existing file with an Open dialog.

Arquivos Recentes Open a list of recent files. Click one to open it.

Open Module… Open an existing module (searches sys.path).

Class Browser Show functions, classes, and methods in the current Editor file in a tree structure. In the shell, open a module first.

Path Browser Show sys.path directories, modules, functions, classes and methods in a tree structure.

Salvar Save the current window to the associated file, if there is one. Windows that have been changed since being opened or last saved have a * before and after the window title. If there is no associated file, do Save As instead.

Salvar como… Save the current window with a Save As dialog. The file saved becomes the new associated file for the window.

Save Copy As… Save the current window to different file without changing the associated file.

Print Window Print the current window to the default printer.

Close Close the current window (ask to save if unsaved).

Exit Close all windows and quit IDLE (ask to save unsaved windows).

Edit menu (Shell and Editor)

Desfazer Desfaz a última alteração na janela atual. Um máximo de 1000 alterações podem ser desfeitas.

Redo Redo the last undone change to the current window.

Cut Copy selection into the system-wide clipboard; then delete the selection.

Copy Copy selection into the system-wide clipboard.

Paste Insert contents of the system-wide clipboard into the current window.

The clipboard functions are also available in context menus.

Select All Select the entire contents of the current window.

Find… Open a search dialog with many options

Find Again Repeat the last search, if there is one.

Find Selection Search for the currently selected string, if there is one.

Find in Files… Open a file search dialog. Put results in a new output window.

Replace… Open a search-and-replace dialog.

Go to Line Move the cursor to the beginning of the line requested and make that line visible. A request past the end of the file goes to the end. Clear any selection and update the line and column status.

Show Completions Open a scrollable list allowing selection of keywords and attributes. See *Completions* in the Editing and navigation section below.

Expand Word Expand a prefix you have typed to match a full word in the same window; repeat to get a different expansion.

Show call tip After an unclosed parenthesis for a function, open a small window with function parameter hints. See *Calltips* in the Editing and navigation section below.

Show surrounding parens Highlight the surrounding parenthesis.

Format menu (Editor window only)

Indent Region Shift selected lines right by the indent width (default 4 spaces).

Dedent Region Shift selected lines left by the indent width (default 4 spaces).

Comment Out Region Insert ## in front of selected lines.

Uncomment Region Remove leading # or ## from selected lines.

Tabify Region Turn *leading* stretches of spaces into tabs. (Note: We recommend using 4 space blocks to indent Python code.)

Untabify Region Turn *all* tabs into the correct number of spaces.

Toggle Tabs Open a dialog to switch between indenting with spaces and tabs.

New Indent Width Open a dialog to change indent width. The accepted default by the Python community is 4 spaces.

Format Paragraph Reformat the current blank-line-delimited paragraph in comment block or multiline string or selected line in a string. All lines in the paragraph will be formatted to less than N columns, where N defaults to 72.

Strip trailing whitespace Remove trailing space and other whitespace characters after the last non-whitespace character of a line by applying str.rstrip to each line, including lines within multiline strings. Except for Shell windows, remove extra newlines at the end of the file.

Run menu (Editor window only)

Run Module Do *Check Module*. If no error, restart the shell to clean the environment, then execute the module. Output is displayed in the Shell window. Note that output requires use of `print` or `write`. When execution is complete, the Shell retains focus and displays a prompt. At this point, one may interactively explore the result of execution. This is similar to executing a file with `python -i file` at a command line.

Run… Customized Same as *Run Module*, but run the module with customized settings. *Command Line Arguments* extend `sys.argv` as if passed on a command line. The module can be run in the Shell without restarting.

Check Module Check the syntax of the module currently open in the Editor window. If the module has not been saved IDLE will either prompt the user to save or autosave, as selected in the General tab of the Idle Settings dialog. If there is a syntax error, the approximate location is indicated in the Editor window.

Python Shell Open or wake up the Python Shell window.

Shell menu (Shell window only)

View Last Restart Scroll the shell window to the last Shell restart.

Restart Shell Restart the shell to clean the environment.

Previous History Cycle through earlier commands in history which match the current entry.

Next History Cycle through later commands in history which match the current entry.

Interrupt Execution Stop a running program.

Debug menu (Shell window only)

Go to File/Line Look on the current line. with the cursor, and the line above for a filename and line number. If found, open the file if not already open, and show the line. Use this to view source lines referenced in an exception traceback and lines found by Find in Files. Also available in the context menu of the Shell window and Output windows.

Debugger (toggle) When activated, code entered in the Shell or run from an Editor will run under the debugger. In the Editor, breakpoints can be set with the context menu. This feature is still incomplete and somewhat experimental.

Stack Viewer Show the stack traceback of the last exception in a tree widget, with access to locals and globals.

Auto-open Stack Viewer Toggle automatically opening the stack viewer on an unhandled exception.

Options menu (Shell and Editor)

Configure IDLE Open a configuration dialog and change preferences for the following: fonts, indentation, keybindings, text color themes, startup windows and size, additional help sources, and extensions. On macOS, open the configuration dialog by selecting Preferences in the application menu. For more details, see *Setting preferences* under Help and preferences.

Most configuration options apply to all windows or all future windows. The option items below only apply to the active window.

Show/Hide Code Context (Editor Window only) Open a pane at the top of the edit window which shows the block context of the code which has scrolled above the top of the window. See *Code Context* in the Editing and Navigation section below.

Show/Hide Line Numbers (Editor Window only) Open a column to the left of the edit window which shows the number of each line of text. The default is off, which may be changed in the preferences (see *Setting preferences*).

Zoom/Restore Height Toggles the window between normal size and maximum height. The initial size defaults to 40 lines by 80 chars unless changed on the General tab of the Configure IDLE dialog. The maximum height for a screen is determined by momentarily maximizing a window the first time one is zoomed on the screen. Changing screen settings may invalidate the saved height. This toggle has no effect when a window is maximized.

Window menu (Shell and Editor)

Lists the names of all open windows; select one to bring it to the foreground (deiconifying it if necessary).

Help menu (Shell and Editor)

About IDLE Display version, copyright, license, credits, and more.

IDLE Help Display this IDLE document, detailing the menu options, basic editing and navigation, and other tips.

Documentação do Python Access local Python documentation, if installed, or start a web browser and open docs.python.org showing the latest Python documentation.

Demonstração com o Turtle Run the turtledemo module with example Python code and turtle drawings.

Additional help sources may be added here with the Configure IDLE dialog under the General tab. See the *Help sources* subsection below for more on Help menu choices.

Context Menus

Open a context menu by right-clicking in a window (Control-click on macOS). Context menus have the standard clipboard functions also on the Edit menu.

Cut Copy selection into the system-wide clipboard; then delete the selection.

Copy Copy selection into the system-wide clipboard.

Paste Insert contents of the system-wide clipboard into the current window.

Editor windows also have breakpoint functions. Lines with a breakpoint set are specially marked. Breakpoints only have an effect when running under the debugger. Breakpoints for a file are saved in the user's `.idlerc` directory.

Set Breakpoint Set a breakpoint on the current line.

Clear Breakpoint Clear the breakpoint on that line.

Shell and Output windows also have the following.

Go to file/line Same as in Debug menu.

The Shell window also has an output squeezing facility explained in the *Python Shell window* subsection below.

Squeeze If the cursor is over an output line, squeeze all the output between the code above and the prompt below down to a 'Squeezed text' label.

26.5.2 Editing and navigation

Editor windows

IDLE may open editor windows when it starts, depending on settings and how you start IDLE. Thereafter, use the File menu. There can be only one open editor window for a given file.

The title bar contains the name of the file, the full path, and the version of Python and IDLE running the window. The status bar contains the line number ('Ln') and column number ('Col'). Line numbers start with 1; column numbers with 0.

IDLE assumes that files with a known .py* extension contain Python code and that other files do not. Run Python code with the Run menu.

Teclas de atalho

In this section, 'C' refers to the `Control` key on Windows and Unix and the `Command` key on macOS.

- `Backspace` deletes to the left; `Del` deletes to the right
- `C-Backspace` delete word left; `C-Del` delete word to the right
- Arrow keys and `Page Up`/`Page Down` to move around
- `C-LeftArrow` and `C-RightArrow` moves by words
- `Home`/`End` go to begin/end of line
- `C-Home`/`C-End` go to begin/end of file
- Some useful Emacs bindings are inherited from Tcl/Tk:
  - `C-a` beginning of line
  - `C-e` end of line
  - `C-k` kill line (but doesn't put it in clipboard)
  - `C-l` center window around the insertion point
  - `C-b` go backward one character without deleting (usually you can also use the cursor key for this)
  - `C-f` go forward one character without deleting (usually you can also use the cursor key for this)
  - `C-p` go up one line (usually you can also use the cursor key for this)
  - `C-d` delete next character

Standard keybindings (like `C-c` to copy and `C-v` to paste) may work. Keybindings are selected in the Configure IDLE dialog.

Automatic indentation

After a block-opening statement, the next line is indented by 4 spaces (in the Python Shell window by one tab). After certain keywords (break, return etc.) the next line is dedented. In leading indentation, `Backspace` deletes up to 4 spaces if they are there. `Tab` inserts spaces (in the Python Shell window one tab), number depends on Indent width. Currently, tabs are restricted to four spaces due to Tcl/Tk limitations.

See also the indent/dedent region commands on the *Format menu*.

Completions

Completions are supplied for functions, classes, and attributes of classes, both built-in and user-defined. Completions are also provided for filenames.

The AutoCompleteWindow (ACW) will open after a predefined delay (default is two seconds) after a '.' or (in a string) an os.sep is typed. If after one of those characters (plus zero or more other characters) a tab is typed the ACW will open immediately if a possible continuation is found.

If there is only one possible completion for the characters entered, a `Tab` will supply that completion without opening the ACW.

'Show Completions' will force open a completions window, by default the `C-space` will open a completions window. In an empty string, this will contain the files in the current directory. On a blank line, it will contain the built-in and user-defined functions and classes in the current namespaces, plus any modules imported. If some characters have been entered, the ACW will attempt to be more specific.

If a string of characters is typed, the ACW selection will jump to the entry most closely matching those characters. Entering a `tab` will cause the longest non-ambiguous match to be entered in the Editor window or Shell. Two `tab` in a row will supply the current ACW selection, as will return or a double click. Cursor keys, Page Up/Down, mouse selection, and the scroll wheel all operate on the ACW.

"Hidden" attributes can be accessed by typing the beginning of hidden name after a '.', e.g. '_'. This allows access to modules with `__all__` set, or to class-private attributes.

Completions and the 'Expand Word' facility can save a lot of typing!

Completions are currently limited to those in the namespaces. Names in an Editor window which are not via `__main__` and *`sys.modules`* will not be found. Run the module once with your imports to correct this situation. Note that IDLE itself places quite a few modules in sys.modules, so much can be found by default, e.g. the re module.

If you don't like the ACW popping up unbidden, simply make the delay longer or disable the extension.

Calltips

A calltip is shown when one types `(` after the name of an *accessible* function. A name expression may include dots and subscripts. A calltip remains until it is clicked, the cursor is moved out of the argument area, or `)` is typed. When the cursor is in the argument part of a definition, the menu or shortcut display a calltip.

A calltip consists of the function signature and the first line of the docstring. For builtins without an accessible signature, the calltip consists of all lines up the fifth line or the first blank line. These details may change.

The set of *accessible* functions depends on what modules have been imported into the user process, including those imported by Idle itself, and what definitions have been run, all since the last restart.

For example, restart the Shell and enter `itertools.count(`. A calltip appears because Idle imports itertools into the user process for its own use. (This could change.) Enter `turtle.write(` and nothing appears. Idle does not import turtle. The menu or shortcut do nothing either. Enter `import turtle` and then `turtle.write(` will work.

In an editor, import statements have no effect until one runs the file. One might want to run a file after writing the import statements at the top, or immediately run an existing file before editing.

Code Context

Within an editor window containing Python code, code context can be toggled in order to show or hide a pane at the top of the window. When shown, this pane freezes the opening lines for block code, such as those beginning with `class`, `def`, or `if` keywords, that would have otherwise scrolled out of view. The size of the pane will be expanded and contracted as needed to show the all current levels of context, up to the maximum number of lines defined in the Configure IDLE dialog (which defaults to 15). If there are no current context lines and the feature is toggled on, a single blank line will display. Clicking on a line in the context pane will move that line to the top of the editor.

The text and background colors for the context pane can be configured under the Highlights tab in the Configure IDLE dialog.

Python Shell window

With IDLE's Shell, one enters, edits, and recalls complete statements. Most consoles and terminals only work with a single physical line at a time.

When one pastes code into Shell, it is not compiled and possibly executed until one hits `Return`. One may edit pasted code first. If one pastes more that one statement into Shell, the result will be a *`SyntaxError`* when multiple statements are compiled as if they were one.

The editing features described in previous subsections work when entering code interactively. IDLE's Shell window also responds to the following keys.

- `C-c` interrupts executing command
- `C-d` sends end-of-file; closes window if typed at a >>> prompt
- `Alt-/` (Expand word) is also useful to reduce typing

  Command history

  - `Alt-p` retrieves previous command matching what you have typed. On macOS use `C-p`.
  - `Alt-n` retrieves next. On macOS use `C-n`.
  - `Return` while on any previous command retrieves that command

Text colors

Idle defaults to black on white text, but colors text with special meanings. For the shell, these are shell output, shell error, user output, and user error. For Python code, at the shell prompt or in an editor, these are keywords, builtin class and function names, names following `class` and `def`, strings, and comments. For any text window, these are the cursor (when present), found text (when possible), and selected text.

Text coloring is done in the background, so uncolorized text is occasionally visible. To change the color scheme, use the Configure IDLE dialog Highlighting tab. The marking of debugger breakpoint lines in the editor and text in popups and dialogs is not user-configurable.

26.5.3 Startup and code execution

Upon startup with the `-s` option, IDLE will execute the file referenced by the environment variables `IDLESTARTUP` or `PYTHONSTARTUP`. IDLE first checks for `IDLESTARTUP`; if `IDLESTARTUP` is present the file referenced is run. If `IDLESTARTUP` is not present, IDLE checks for `PYTHONSTARTUP`. Files referenced by these environment variables are convenient places to store functions that are used frequently from the IDLE shell, or for executing import statements to import common modules.

In addition, `Tk` also loads a startup file if it is present. Note that the Tk file is loaded unconditionally. This additional file is `.Idle.py` and is looked for in the user's home directory. Statements in this file will be executed in the Tk namespace, so this file is not useful for importing functions to be used from IDLE's Python shell.

Uso na linha de comando

```
idle.py [-c command] [-d] [-e] [-h] [-i] [-r file] [-s] [-t title] [-] [arg] ...

-c command  run command in the shell window
-d          enable debugger and open shell window
-e          open editor window
-h          print help message with legal combinations and exit
-i          open shell window
-r file     run file in shell window
-s          run $IDLESTARTUP or $PYTHONSTARTUP first, in shell window
-t title    set title of shell window
-           run stdin in shell (- must be last option before args)
```

If there are arguments:

- If `-`, `-c`, or `r` is used, all arguments are placed in `sys.argv[1:...]` and `sys.argv[0]` is set to `''`, `'-c'`, or `'-r'`. No editor window is opened, even if that is the default set in the Options dialog.
- Otherwise, arguments are files opened for editing and `sys.argv` reflects the arguments passed to IDLE itself.

Startup failure

IDLE uses a socket to communicate between the IDLE GUI process and the user code execution process. A connection must be established whenever the Shell starts or restarts. (The latter is indicated by a divider line that says 'RESTART"). If the user process fails to connect to the GUI process, it displays a `Tk` error box with a 'cannot connect' message that directs the user here. It then exits.

A common cause of failure is a user-written file with the same name as a standard library module, such as *random.py* and *tkinter.py*. When such a file is located in the same directory as a file that is about to be run, IDLE cannot import the stdlib file. The current fix is to rename the user file.

Though less common than in the past, an antivirus or firewall program may stop the connection. If the program cannot be taught to allow the connection, then it must be turned off for IDLE to work. It is safe to allow this internal connection because no data is visible on external ports. A similar problem is a network mis-configuration that blocks connections.

Python installation issues occasionally stop IDLE: multiple versions can clash, or a single installation might need admin access. If one undo the clash, or cannot or does not want to run as admin, it might be easiest to completely remove Python and start over.

A zombie pythonw.exe process could be a problem. On Windows, use Task Manager to check for one and stop it if there is. Sometimes a restart initiated by a program crash or Keyboard Interrupt (control-C) may fail to connect. Dismissing the error box or using Restart Shell on the Shell menu may fix a temporary problem.

When IDLE first starts, it attempts to read user configuration files in `~/.idlerc/` (~ is one's home directory). If there is a problem, an error message should be displayed. Leaving aside random disk glitches, this can be prevented by never editing the files by hand. Instead, use the configuration dialog, under Options. Once there is an error in a user configuration file, the best solution may be to delete it and start over with the settings dialog.

If IDLE quits with no message, and it was not started from a console, try starting it from a console or terminal (`python -m idlelib`) and see if this results in an error message.

Running user code

With rare exceptions, the result of executing Python code with IDLE is intended to be the same as executing the same code by the default method, directly with Python in a text-mode system console or terminal window. However, the different interface and operation occasionally affect visible results. For instance, `sys.modules` starts with more entries, and `threading.activeCount()` returns 2 instead of 1.

By default, IDLE runs user code in a separate OS process rather than in the user interface process that runs the shell and editor. In the execution process, it replaces `sys.stdin`, `sys.stdout`, and `sys.stderr` with objects that get input from and send output to the Shell window. The original values stored in `sys.__stdin__`, `sys.__stdout__`, and `sys.__stderr__` are not touched, but may be `None`.

When Shell has the focus, it controls the keyboard and screen. This is normally transparent, but functions that directly access the keyboard and screen will not work. These include system-specific functions that determine whether a key has been pressed and if so, which.

IDLE's standard stream replacements are not inherited by subprocesses created in the execution process, whether directly by user code or by modules such as multiprocessing. If such subprocess use `input` from sys.stdin or `print` or `write` to sys.stdout or sys.stderr, IDLE should be started in a command line window. The secondary subprocess will then be attached to that window for input and output.

The IDLE code running in the execution process adds frames to the call stack that would not be there otherwise. IDLE wraps `sys.getrecursionlimit` and `sys.setrecursionlimit` to reduce the effect of the additional stack frames.

If `sys` is reset by user code, such as with `importlib.reload(sys)`, IDLE's changes are lost and input from the keyboard and output to the screen will not work correctly.

When user code raises SystemExit either directly or by calling sys.exit, IDLE returns to a Shell prompt instead of exiting.

User output in Shell

When a program outputs text, the result is determined by the corresponding output device. When IDLE executes user code, `sys.stdout` and `sys.stderr` are connected to the display area of IDLE's Shell. Some of its features are inherited from the underlying Tk Text widget. Others are programmed additions. Where it matters, Shell is designed for development rather than production runs.

For instance, Shell never throws away output. A program that sends unlimited output to Shell will eventually fill memory, resulting in a memory error. In contrast, some system text windows only keep the last n lines of output. A Windows console, for instance, keeps a user-settable 1 to 9999 lines, with 300 the default.

A Tk Text widget, and hence IDLE's Shell, displays characters (codepoints) in the BMP (Basic Multilingual Plane) subset of Unicode. Which characters are displayed with a proper glyph and which with a replacement box depends on the operating system and installed fonts. Tab characters cause the following text to begin after the next tab stop. (They occur every 8 'characters'). Newline characters cause following text to appear on a new line. Other control characters are ignored or displayed as a space, box, or something else, depending on the operating system and font. (Moving the text cursor through such output with arrow keys may exhibit some surprising spacing behavior.)

```
>>> s = 'a\tb\a<\x02><\r>\bc\nd'  # Enter 22 chars.
>>> len(s)
14
>>> s  # Display repr(s)
'a\tb\x07<\x02><\r>\x08c\nd'
>>> print(s, end='')  # Display s as is.
# Result varies by OS and font.  Try it.
```

The `repr` function is used for interactive echo of expression values. It returns an altered version of the input string in which control codes, some BMP codepoints, and all non-BMP codepoints are replaced with escape codes. As demonstrated above, it allows one to identify the characters in a string, regardless of how they are displayed.

Normal and error output are generally kept separate (on separate lines) from code input and each other. They each get different highlight colors.

For SyntaxError tracebacks, the normal '^' marking where the error was detected is replaced by coloring the text with an error highlight. When code run from a file causes other exceptions, one may right click on a traceback line to jump to the corresponding line in an IDLE editor. The file will be opened if necessary.

Shell has a special facility for squeezing output lines down to a 'Squeezed text' label. This is done automatically for output over N lines (N = 50 by default). N can be changed in the PyShell section of the General page of the Settings dialog. Output with fewer lines can be squeezed by right clicking on the output. This can be useful lines long enough to slow down scrolling.

Squeezed output is expanded in place by double-clicking the label. It can also be sent to the clipboard or a separate view window by right-clicking the label.

Developing tkinter applications

IDLE is intentionally different from standard Python in order to facilitate development of tkinter programs. Enter `import tkinter as tk; root = tk.Tk()` in standard Python and nothing appears. Enter the same in IDLE and a tk window appears. In standard Python, one must also enter `root.update()` to see the window. IDLE does the equivalent in the background, about 20 times a second, which is about every 50 milliseconds. Next enter `b = tk.Button(root, text='button'); b.pack()`. Again, nothing visibly changes in standard Python until one enters `root.update()`.

Most tkinter programs run `root.mainloop()`, which usually does not return until the tk app is destroyed. If the program is run with `python -i` or from an IDLE editor, a `>>>` shell prompt does not appear until `mainloop()` returns, at which time there is nothing left to interact with.

When running a tkinter program from an IDLE editor, one can comment out the mainloop call. One then gets a shell prompt immediately and can interact with the live application. One just has to remember to re-enable the mainloop call when running in standard Python.

Running without a subprocess

By default, IDLE executes user code in a separate subprocess via a socket, which uses the internal loopback interface. This connection is not externally visible and no data is sent to or received from the Internet. If firewall software complains anyway, you can ignore it.

If the attempt to make the socket connection fails, Idle will notify you. Such failures are sometimes transient, but if persistent, the problem may be either a firewall blocking the connection or misconfiguration of a particular system. Until the problem is fixed, one can run Idle with the -n command line switch.

If IDLE is started with the -n command line switch it will run in a single process and will not create the subprocess which runs the RPC Python execution server. This can be useful if Python cannot create the subprocess or the RPC socket interface on your platform. However, in this mode user code is not isolated from IDLE itself. Also, the environment is not restarted when Run/Run Module (F5) is selected. If your code has been modified, you must reload() the affected modules and re-import any specific items (e.g. from foo import baz) if the changes are to take effect. For these reasons, it is preferable to run IDLE with the default subprocess if at all possible.

Obsoleto desde a versão 3.4.

26.5.4 Help and preferences

Help sources

Help menu entry "IDLE Help" displays a formatted html version of the IDLE chapter of the Library Reference. The result, in a read-only tkinter text window, is close to what one sees in a web browser. Navigate through the text with a mousewheel, the scrollbar, or up and down arrow keys held down. Or click the TOC (Table of Contents) button and select a section header in the opened box.

Help menu entry "Python Docs" opens the extensive sources of help, including tutorials, available at `docs.python.org/x.y`, where 'x.y' is the currently running Python version. If your system has an off-line copy of the docs (this may be an installation option), that will be opened instead.

Selected URLs can be added or removed from the help menu at any time using the General tab of the Configure IDLE dialog.

Setting preferences

The font preferences, highlighting, keys, and general preferences can be changed via Configure IDLE on the Option menu. Non-default user settings are saved in a `.idlerc` directory in the user's home directory. Problems caused by bad user configuration files are solved by editing or deleting one or more of the files in `.idlerc`.

On the Font tab, see the text sample for the effect of font face and size on multiple characters in multiple languages. Edit the sample to add other characters of personal interest. Use the sample to select monospaced fonts. If particular characters have problems in Shell or an editor, add them to the top of the sample and try changing first size and then font.

On the Highlights and Keys tab, select a built-in or custom color theme and key set. To use a newer built-in color theme or key set with older IDLEs, save it as a new custom theme or key set and it well be accessible to older IDLEs.

IDLE on macOS

Under System Preferences: Dock, one can set "Prefer tabs when opening documents" to "Always". This setting is not compatible with the tk/tkinter GUI framework used by IDLE, and it breaks a few IDLE features.

Extensions

IDLE contains an extension facility. Preferences for extensions can be changed with the Extensions tab of the preferences dialog. See the beginning of config-extensions.def in the idlelib directory for further information. The only current default extension is zzdummy, an example also used for testing.

26.6 Outros Pacotes de Interface Gráficas de Usuário

Major cross-platform (Windows, Mac OS X, Unix-like) GUI toolkits are available for Python:

Ver também:

PyGObject PyGObject provides introspection bindings for C libraries using GObject. One of these libraries is the GTK+ 3 widget set. GTK+ comes with many more widgets than Tkinter provides. An online Python GTK+ 3 Tutorial is available.

PyGTK PyGTK provides bindings for an older version of the library, GTK+ 2. It provides an object oriented interface that is slightly higher level than the C one. There are also bindings to GNOME. An online tutorial is available.

PyQt PyQt is a **sip**-wrapped binding to the Qt toolkit. Qt is an extensive C++ GUI application development framework that is available for Unix, Windows and Mac OS X. **sip** is a tool for generating bindings for C++ libraries as Python classes, and is specifically designed for Python.

PySide2 Also known as the Qt for Python project, PySide2 is a newer binding to the Qt toolkit. It is provided by The Qt Company and aims to provide a complete port of PySide to Qt 5. Compared to PyQt, its licensing scheme is friendlier to non-open source applications.

wxPython wxPython is a cross-platform GUI toolkit for Python that is built around the popular wxWidgets (formerly wxWindows) C++ toolkit. It provides a native look and feel for applications on Windows, Mac OS X, and Unix systems by using each platform's native widgets where ever possible, (GTK+ on Unix-like systems). In addition to an extensive set of widgets, wxPython provides classes for online documentation and context sensitive help, printing, HTML viewing, low-level device context drawing, drag and drop, system clipboard access, an XML-based resource format and more, including an ever growing library of user-contributed modules.

PyGTK, PyQt, PySide2, and wxPython, all have a modern look and feel and more widgets than Tkinter. In addition, there are many other GUI toolkits for Python, both cross-platform, and platform-specific. See the GUI Programming page in the Python Wiki for a much more complete list, and also for links to documents where the different GUI toolkits are compared.

CAPÍTULO 27

Ferramentas de Desenvolvimento

Os módulos descritos neste capítulo ajudam você a escrever softwares. Por exemplo, o módulo *pydoc* recebe um módulo e gera documentação com base no conteúdo do módulo. Os módulos *doctest* e *unittest* contêm frameworks para escrever testes unitários que automaticamente exercitam código e verificam se a saída esperada é produzida. **2to3** pode traduzir o código-fonte do Python 2.x para um código válido do Python 3.x.

A lista de módulos descritos neste capítulo é:

27.1 typing — Suporte para dicas de tipo

Novo na versão 3.5.

Código Fonte: Lib/typing.py

Nota: O tempo de execução do Python não força anotações de tipos de variáveis e funções. Elas podem ser usadas por ferramentas de terceiros como verificadores de tipo, IDEs, linters, etc.

This module supports type hints as specified by **PEP 484** and **PEP 526**. The most fundamental support consists of the types *Any*, *Union*, *Tuple*, *Callable*, *TypeVar*, and *Generic*. For full specification please see **PEP 484**. For a simplified introduction to type hints see **PEP 483**.

A função abaixo recebe e retorna uma string e é anotada como a seguir:

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

Na função `greeting`, é esperado que o argumento `name` seja do tipo *str* e o retorno do tipo *str*. Subtipos são aceitos como argumentos.

27.1.1 Apelidos de tipo

A type alias is defined by assigning the type to the alias. In this example, `Vector` and `List[float]` will be treated as interchangeable synonyms:

```
from typing import List
Vector = List[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# typechecks; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

Apelidos de tipo são úteis para simplificar assinaturas de tipo complexas. Por exemplo:

```
from typing import Dict, Tuple, Sequence

ConnectionOptions = Dict[str, str]
Address = Tuple[str, int]
Server = Tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: Sequence[Server]) -> None:
    ...

# The static type checker will treat the previous type signature as
# being exactly equivalent to this one.
def broadcast_message(
        message: str,
        servers: Sequence[Tuple[Tuple[str, int], Dict[str, str]]]) -> None:
    ...
```

Note que `None` como uma dica de tipo é um caso especial e é substituído por `type(None)`.

27.1.2 NewType

Use the *`NewType()`* helper function to create distinct types:

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

O verificador de tipo estático tratará o novo tipo como se fosse uma subclasse do tipo original. Isso é útil para ajudar a encontrar erros de lógica:

```
def get_user_name(user_id: UserId) -> str:
    ...

# typechecks
user_a = get_user_name(UserId(42351))

# does not typecheck; an int is not a UserId
user_b = get_user_name(-1)
```

Você ainda pode executar todas as operações `int` em uma variável do tipo `UserId`, mas o resultado sempre será do tipo `int`. Isso permite que você passe um `UserId` em qualquer ocasião que `int` possa ser esperado, mas previne que

você acidentalmente crie um `UserId` de uma forma inválida:

```
# 'output' is of type 'int', not 'UserId'
output = UserId(23413) + UserId(54341)
```

Note that these checks are enforced only by the static type checker. At runtime the statement `Derived = NewType('Derived', Base)` will make `Derived` a function that immediately returns whatever parameter you pass it. That means the expression `Derived(some_value)` does not create a new class or introduce any overhead beyond that of a regular function call.

Mais precisamente, a expressão `some_value is Derived(some_value)` é sempre verdadeira em tempo de execução.

This also means that it is not possible to create a subtype of `Derived` since it is an identity function at runtime, not an actual type:

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not typecheck
class AdminUserId(UserId): pass
```

However, it is possible to create a *`NewType()`* based on a 'derived' `NewType`:

```
from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)
```

e a verificação de tipo para `ProUserId` funcionará como esperado.

Veja **PEP 484** para mais detalhes.

Nota: Relembre que o uso de um apelido de tipo declara que dois tipos serão *equivalentes* entre si. Efetuar `Alias = Original` irá fazer o verificador de tipo estático tratar `Alias` como sendo *exatamente equivalente* a `Original` em todos os casos. Isso é útil quando você deseja simplificar assinaturas de tipo complexas.

Em contraste, `NewType` declara que um tipo será *subtipo* de outro. Efetuando `Derived = NewType('Derived', Original)` irá fazer o verificador de tipo estático tratar `Derived` como uma *subclasse* de `Original`, o que significa que um valor do tipo `Original` não pode ser utilizado onde um valor do tipo `Derived` é esperado. Isso é útil quando você deseja evitar erros de lógica com custo mínimo de tempo de execução.

Novo na versão 3.5.2.

27.1.3 Callable

Frameworks que esperam funções de retorno com assinaturas específicas podem ter seus tipos indicados usando"Callable[[Arg1Type, Arg2Type], ReturnType]".

Por exemplo:

```
from typing import Callable

def feeder(get_next_item: Callable[[], str]) -> None:
```

(continua na próxima página)

(continuação da página anterior)

```
    # Body

def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None]) -> None:
    # Body
```

É possível declarar o tipo de retorno de um chamável sem especificar a assinatura da chamada, substituindo por reticências literais a lista de argumentos na dica de tipo: `Callable[..., ReturnType]`.

27.1.4 Generics

Since type information about objects kept in containers cannot be statically inferred in a generic way, abstract base classes have been extended to support subscription to denote expected types for container elements.

```
from typing import Mapping, Sequence

def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None: ...
```

Generics can be parameterized by using a new factory available in typing called *TypeVar*.

```
from typing import Sequence, TypeVar

T = TypeVar('T')      # Declare type variable

def first(l: Sequence[T]) -> T:   # Generic function
    return l[0]
```

27.1.5 User-defined generic types

A user-defined class can be defined as a generic class.

```
from typing import TypeVar, Generic
from logging import Logger

T = TypeVar('T')

class LoggedVar(Generic[T]):
    def __init__(self, value: T, name: str, logger: Logger) -> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

`Generic[T]` as a base class defines that the class `LoggedVar` takes a single type parameter `T` . This also makes `T` valid as a type within the class body.

The *Generic* base class defines `__class_getitem__()` so that `LoggedVar[t]` is valid as a type:

```
from typing import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

A generic type can have any number of type variables, and type variables may be constrained:

```
from typing import TypeVar, Generic
...

T = TypeVar('T')
S = TypeVar('S', int, str)

class StrangePair(Generic[T, S]):
    ...
```

Each type variable argument to *Generic* must be distinct. This is thus invalid:

```
from typing import TypeVar, Generic
...

T = TypeVar('T')

class Pair(Generic[T, T]):   # INVALID
    ...
```

You can use multiple inheritance with *Generic*:

```
from typing import TypeVar, Generic, Sized

T = TypeVar('T')

class LinkedList(Sized, Generic[T]):
    ...
```

When inheriting from generic classes, some type variables could be fixed:

```
from typing import TypeVar, Mapping

T = TypeVar('T')

class MyDict(Mapping[str, T]):
    ...
```

In this case `MyDict` has a single parameter, `T`.

Using a generic class without specifying type parameters assumes *Any* for each position. In the following example, `MyIterable` is not generic but implicitly inherits from `Iterable[Any]`:

```
from typing import Iterable

class MyIterable(Iterable): # Same as Iterable[Any]
```

User defined generic type aliases are also supported. Examples:

```
from typing import TypeVar, Iterable, Tuple, Union
S = TypeVar('S')
Response = Union[Iterable[S], int]

# Return type here is same as Union[Iterable[str], int]
def response(query: str) -> Response[str]:
    ...

T = TypeVar('T', int, float, complex)
Vec = Iterable[Tuple[T, T]]

def inproduct(v: Vec[T]) -> T: # Same as Iterable[Tuple[T, T]]
    return sum(x*y for x, y in v)
```

Alterado na versão 3.7: *Generic* no longer has a custom metaclass.

A user-defined generic class can have ABCs as base classes without a metaclass conflict. Generic metaclasses are not supported. The outcome of parameterizing generics is cached, and most types in the typing module are hashable and comparable for equality.

27.1.6 The `Any` type

A special kind of type is *Any*. A static type checker will treat every type as being compatible with *Any* and *Any* as being compatible with every type.

This means that it is possible to perform any operation or method call on a value of type on *Any* and assign it to any variable:

```
from typing import Any

a = None     # type: Any
a = []       # OK
a = 2        # OK

s = ''       # type: str
s = a        # OK

def foo(item: Any) -> int:
    # Typechecks; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

Notice that no typechecking is performed when assigning a value of type *Any* to a more precise type. For example, the static type checker did not report an error when assigning `a` to `s` even though `s` was declared to be of type *str* and receives an *int* value at runtime!

Furthermore, all functions without a return type or parameter types will implicitly default to using *Any*:

```
def legacy_parser(text):
    ...
    return data

# A static type checker will treat the above
# as having the same signature as:
```

(continua na próxima página)

(continuação da página anterior)

```
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

This behavior allows *Any* to be used as an *escape hatch* when you need to mix dynamically and statically typed code.

Contrast the behavior of *Any* with the behavior of *object*. Similar to *Any*, every type is a subtype of *object*. However, unlike *Any*, the reverse is not true: *object* is *not* a subtype of every other type.

That means when the type of a value is *object*, a type checker will reject almost all operations on it, and assigning it to a variable (or using it as a return value) of a more specialized type is a type error. For example:

```
def hash_a(item: object) -> int:
    # Fails; an object does not have a 'magic' method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Typechecks
    item.magic()
    ...

# Typechecks, since ints and strs are subclasses of object
hash_a(42)
hash_a("foo")

# Typechecks, since Any is compatible with all types
hash_b(42)
hash_b("foo")
```

Use *object* to indicate that a value could be any type in a typesafe manner. Use *Any* to indicate that a value is dynamically typed.

27.1.7 Classes, functions, and decorators

The module defines the following classes, functions and decorators:

class typing.**TypeVar**

Tipo variável.

Utilização:

```
T = TypeVar('T')  # Can be anything
A = TypeVar('A', str, bytes)  # Must be str or bytes
```

Type variables exist primarily for the benefit of static type checkers. They serve as the parameters for generic types as well as for generic function definitions. See class Generic for more information on generic types. Generic functions work as follows:

```
def repeat(x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
    return [x]*n

def longest(x: A, y: A) -> A:
    """Return the longest of two strings."""
    return x if len(x) >= len(y) else y
```

The latter example's signature is essentially the overloading of `(str, str) -> str` and `(bytes, bytes) -> bytes`. Also note that if the arguments are instances of some subclass of *str*, the return type is still plain *str*.

At runtime, `isinstance(x, T)` will raise *TypeError*. In general, *isinstance()* and *issubclass()* should not be used with types.

Type variables may be marked covariant or contravariant by passing `covariant=True` or `contravariant=True`. See **PEP 484** for more details. By default type variables are invariant. Alternatively, a type variable may specify an upper bound using `bound=<type>`. This means that an actual type substituted (explicitly or implicitly) for the type variable must be a subclass of the boundary type, see **PEP 484**.

class typing.**Generic**

Abstract base class for generic types.

A generic type is typically declared by inheriting from an instantiation of this class with one or more type variables. For example, a generic mapping type might be defined as:

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

This class can then be used as follows:

```
X = TypeVar('X')
Y = TypeVar('Y')

def lookup_name(mapping: Mapping[X, Y], key: X, default: Y) -> Y:
    try:
        return mapping[key]
    except KeyError:
        return default
```

class typing.**Type**(*Generic[CT_co]*)

A variable annotated with `C` may accept a value of type `C`. In contrast, a variable annotated with `Type[C]` may accept values that are classes themselves – specifically, it will accept the *class object* of `C`. For example:

```
a = 3         # Has type 'int'
b = int       # Has type 'Type[int]'
c = type(a)   # Also has type 'Type[int]'
```

Note that `Type[C]` is covariant:

```
class User: ...
class BasicUser(User): ...
class ProUser(User): ...
class TeamUser(User): ...

# Accepts User, BasicUser, ProUser, TeamUser, ...
def make_new_user(user_class: Type[User]) -> User:
    # ...
    return user_class()
```

The fact that `Type[C]` is covariant implies that all subclasses of `C` should implement the same constructor signature and class method signatures as `C`. The type checker should flag violations of this, but should also allow constructor calls in subclasses that match the constructor calls in the indicated base class. How the type checker is required to handle this particular case may change in future revisions of **PEP 484**.

The only legal parameters for *Type* are classes, *Any*, *type variables*, and unions of any of these types. For example:

```
def new_non_team_user(user_class: Type[Union[BaseUser, ProUser]]): ...
```

`Type[Any]` is equivalent to `Type` which in turn is equivalent to `type`, which is the root of Python's metaclass hierarchy.

Novo na versão 3.5.2.

class typing.**Iterable**(*Generic[T_co]*)
A generic version of *collections.abc.Iterable*.

class typing.**Iterator**(*Iterable[T_co]*)
A generic version of *collections.abc.Iterator*.

class typing.**Reversible**(*Iterable[T_co]*)
A generic version of *collections.abc.Reversible*.

class typing.**SupportsInt**
An ABC with one abstract method `__int__`.

class typing.**SupportsFloat**
An ABC with one abstract method `__float__`.

class typing.**SupportsComplex**
An ABC with one abstract method `__complex__`.

class typing.**SupportsBytes**
An ABC with one abstract method `__bytes__`.

class typing.**SupportsAbs**
An ABC with one abstract method `__abs__` that is covariant in its return type.

class typing.**SupportsRound**
An ABC with one abstract method `__round__` that is covariant in its return type.

class typing.**Container**(*Generic[T_co]*)
A generic version of *collections.abc.Container*.

class typing.**Hashable**
An alias to *collections.abc.Hashable*

class typing.**Sized**
An alias to *collections.abc.Sized*

class typing.**Collection**(*Sized, Iterable[T_co], Container[T_co]*)
A generic version of *collections.abc.Collection*

Novo na versão 3.6.0.

class typing.**AbstractSet**(*Sized, Collection[T_co]*)
A generic version of *collections.abc.Set*.

class typing.**MutableSet**(*AbstractSet[T]*)
A generic version of *collections.abc.MutableSet*.

class typing.**Mapping**(*Sized, Collection[KT], Generic[VT_co]*)
A generic version of *collections.abc.Mapping*. This type can be used as follows:

```
def get_position_in_index(word_list: Mapping[str, int], word: str) -> int:
    return word_list[word]
```

class typing.**MutableMapping**(*Mapping[KT, VT]*)
A generic version of *collections.abc.MutableMapping*.

class typing.**Sequence**(*Reversible[T_co], Collection[T_co]*)
A generic version of *collections.abc.Sequence*.

class typing.**MutableSequence**(*Sequence[T]*)
A generic version of *collections.abc.MutableSequence*.

class typing.**ByteString**(*Sequence[int]*)
A generic version of *collections.abc.ByteString*.

This type represents the types *bytes*, *bytearray*, and *memoryview*.

As a shorthand for this type, *bytes* can be used to annotate arguments of any of the types mentioned above.

class typing.**Deque**(*deque, MutableSequence[T]*)
A generic version of *collections.deque*.

Novo na versão 3.5.4.

Novo na versão 3.6.1.

class typing.**List**(*list, MutableSequence[T]*)
Generic version of *list*. Useful for annotating return types. To annotate arguments it is preferred to use an abstract collection type such as *Sequence* or *Iterable*.

This type may be used as follows:

```
T = TypeVar('T', int, float)

def vec2(x: T, y: T) -> List[T]:
    return [x, y]

def keep_positives(vector: Sequence[T]) -> List[T]:
    return [item for item in vector if item > 0]
```

class typing.**Set**(*set, MutableSet[T]*)
A generic version of *builtins.set*. Useful for annotating return types. To annotate arguments it is preferred to use an abstract collection type such as *AbstractSet*.

class typing.**FrozenSet**(*frozenset, AbstractSet[T_co]*)
A generic version of *builtins.frozenset*.

class typing.**MappingView**(*Sized, Iterable[T_co]*)
A generic version of *collections.abc.MappingView*.

class typing.**KeysView**(*MappingView[KT_co], AbstractSet[KT_co]*)
A generic version of *collections.abc.KeysView*.

class typing.**ItemsView**(*MappingView, Generic[KT_co, VT_co]*)
A generic version of *collections.abc.ItemsView*.

class typing.**ValuesView**(*MappingView[VT_co]*)
A generic version of *collections.abc.ValuesView*.

class typing.**Awaitable**(*Generic[T_co]*)
A generic version of *collections.abc.Awaitable*.

Novo na versão 3.5.2.

class typing.**Coroutine**(*Awaitable[V_co], Generic[T_co T_contra, V_co]*)
A generic version of *collections.abc.Coroutine*. The variance and order of type variables correspond to those of *Generator*, for example:

```
from typing import List, Coroutine
c = None # type: Coroutine[List[str], str, int]
...
x = c.send('hi') # type: List[str]
async def bar() -> None:
    x = await c # type: int
```

Novo na versão 3.5.3.

class typing.**AsyncIterable**(*Generic[T_co]*)
Uma versão genérica de *collections.abc.AsyncIterable*.

Novo na versão 3.5.2.

class typing.**AsyncIterator**(*AsyncIterable[T_co]*)
A generic version of *collections.abc.AsyncIterator*.

Novo na versão 3.5.2.

class typing.**ContextManager**(*Generic[T_co]*)
Uma versão genérica de *contextlib.AbstractContextManager*.

Novo na versão 3.5.4.

Novo na versão 3.6.0.

class typing.**AsyncContextManager**(*Generic[T_co]*)
A generic version of *contextlib.AbstractAsyncContextManager*.

Novo na versão 3.5.4.

Novo na versão 3.6.2.

class typing.**Dict**(*dict, MutableMapping[KT, VT]*)
A generic version of *dict*. Useful for annotating return types. To annotate arguments it is preferred to use an abstract collection type such as *Mapping*.

This type can be used as follows:

```
def count_words(text: str) -> Dict[str, int]:
    ...
```

class typing.**DefaultDict**(*collections.defaultdict, MutableMapping[KT, VT]*)
A generic version of *collections.defaultdict*.

Novo na versão 3.5.2.

class typing.**OrderedDict**(*collections.OrderedDict, MutableMapping[KT, VT]*)
A generic version of *collections.OrderedDict*.

Novo na versão 3.7.2.

class typing.**Counter**(*collections.Counter, Dict[T, int]*)
A generic version of *collections.Counter*.

Novo na versão 3.5.4.

Novo na versão 3.6.1.

class typing.**ChainMap**(*collections.ChainMap, MutableMapping[KT, VT]*)
A generic version of *collections.ChainMap*.

Novo na versão 3.5.4.

Novo na versão 3.6.1.

class typing.**Generator**(*Iterator[T_co], Generic[T_co, T_contra, V_co]*)

A generator can be annotated by the generic type `Generator[YieldType, SendType, ReturnType]`. For example:

```
def echo_round() -> Generator[int, float, str]:
    sent = yield 0
    while sent >= 0:
        sent = yield round(sent)
    return 'Done'
```

Note that unlike many other generics in the typing module, the `SendType` of *Generator* behaves contravariantly, not covariantly or invariantly.

If your generator will only yield values, set the `SendType` and `ReturnType` to `None`:

```
def infinite_stream(start: int) -> Generator[int, None, None]:
    while True:
        yield start
        start += 1
```

Alternatively, annotate your generator as having a return type of either `Iterable[YieldType]` or `Iterator[YieldType]`:

```
def infinite_stream(start: int) -> Iterator[int]:
    while True:
        yield start
        start += 1
```

class typing.**AsyncGenerator**(*AsyncIterator[T_co], Generic[T_co, T_contra]*)

An async generator can be annotated by the generic type `AsyncGenerator[YieldType, SendType]`. For example:

```
async def echo_round() -> AsyncGenerator[int, float]:
    sent = yield 0
    while sent >= 0.0:
        rounded = await round(sent)
        sent = yield rounded
```

Unlike normal generators, async generators cannot return a value, so there is no `ReturnType` type parameter. As with *Generator*, the `SendType` behaves contravariantly.

If your generator will only yield values, set the `SendType` to `None`:

```
async def infinite_stream(start: int) -> AsyncGenerator[int, None]:
    while True:
        yield start
        start = await increment(start)
```

Alternatively, annotate your generator as having a return type of either `AsyncIterable[YieldType]` or `AsyncIterator[YieldType]`:

```
async def infinite_stream(start: int) -> AsyncIterator[int]:
    while True:
        yield start
        start = await increment(start)
```

Novo na versão 3.6.1.

class typing.**Text**
Text is an alias for str. It is provided to supply a forward compatible path for Python 2 code: in Python 2, Text is an alias for unicode.

Use Text to indicate that a value must contain a unicode string in a manner that is compatible with both Python 2 and Python 3:

```
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

Novo na versão 3.5.2.

class typing.**IO**
class typing.**TextIO**
class typing.**BinaryIO**
Generic type IO[AnyStr] and its subclasses TextIO(IO[str]) and BinaryIO(IO[bytes]) represent the types of I/O streams such as returned by *open()*.

class typing.**Pattern**
class typing.**Match**
These type aliases correspond to the return types from *re.compile()* and *re.match()*. These types (and the corresponding functions) are generic in AnyStr and can be made specific by writing Pattern[str], Pattern[bytes], Match[str], or Match[bytes].

class typing.**NamedTuple**
Typed version of *collections.namedtuple()*.

Utilização:

```
class Employee(NamedTuple):
    name: str
    id: int
```

This is equivalent to:

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

To give a field a default value, you can assign to it in the class body:

```
class Employee(NamedTuple):
    name: str
    id: int = 3

employee = Employee('Guido')
assert employee.id == 3
```

Fields with a default value must come after any fields without a default.

The resulting class has two extra attributes: _field_types, giving a dict mapping field names to types, and _field_defaults, a dict mapping field names to default values. (The field names are in the _fields attribute, which is part of the namedtuple API.)

NamedTuple subclasses can also have docstrings and methods:

```
class Employee(NamedTuple):
    """Represents an employee."""
    name: str
    id: int = 3
```

(continua na próxima página)

(continuação da página anterior)

```
    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>'
```

Backward-compatible usage:

```
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

Alterado na versão 3.6: Added support for **PEP 526** variable annotation syntax.

Alterado na versão 3.6.1: Added support for default values, methods, and docstrings.

class typing.**ForwardRef**

A class used for internal typing representation of string forward references. For example, `List["SomeClass"]` is implicitly transformed into `List[ForwardRef("SomeClass")]`. This class should not be instantiated by a user, but may be used by introspection tools.

typing.**NewType**(*typ*)

A helper function to indicate a distinct types to a typechecker, see *NewType*. At runtime it returns a function that returns its argument. Usage:

```
UserId = NewType('UserId', int)
first_user = UserId(1)
```

Novo na versão 3.5.2.

typing.**cast**(*typ*, *val*)

Define um valor para um tipo.

This returns the value unchanged. To the type checker this signals that the return value has the designated type, but at runtime we intentionally don't check anything (we want this to be as fast as possible).

typing.**get_type_hints**(*obj*[, *globals*[, *locals*]])

Return a dictionary containing type hints for a function, method, module or class object.

This is often the same as `obj.__annotations__`. In addition, forward references encoded as string literals are handled by evaluating them in `globals` and `locals` namespaces. If necessary, `Optional[t]` is added for function and method annotations if a default value equal to `None` is set. For a class `C`, return a dictionary constructed by merging all the `__annotations__` along `C.__mro__` in reverse order.

@typing.**overload**

The `@overload` decorator allows describing functions and methods that support multiple different combinations of argument types. A series of `@overload`-decorated definitions must be followed by exactly one non-`@overload`-decorated definition (for the same function/method). The `@overload`-decorated definitions are for the benefit of the type checker only, since they will be overwritten by the non-`@overload`-decorated definition, while the latter is used at runtime but should be ignored by a type checker. At runtime, calling a `@overload`-decorated function directly will raise *`NotImplementedError`*. An example of overload that gives a more precise type than can be expressed using a union or a type variable:

```
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> Tuple[int, str]:
    ...
@overload
def process(response: bytes) -> str:
    ...
```

(continua na próxima página)

(continuação da página anterior)

```
def process(response):
    <actual implementation>
```

See **PEP 484** for details and comparison with other typing semantics.

`@typing.`**`no_type_check`**
Decorator to indicate that annotations are not type hints.

This works as class or function *decorator*. With a class, it applies recursively to all methods defined in that class (but not to methods defined in its superclasses or subclasses).

This mutates the function(s) in place.

`@typing.`**`no_type_check_decorator`**
Decorator to give another decorator the *`no_type_check()`* effect.

This wraps the decorator with something that wraps the decorated function in *`no_type_check()`*.

`@typing.`**`type_check_only`**
Decorator to mark a class or function to be unavailable at runtime.

This decorator is itself not available at runtime. It is mainly intended to mark classes that are defined in type stub files if an implementation returns an instance of a private class:

```
@type_check_only
class Response:  # private or not available at runtime
    code: int
    def get_header(self, name: str) -> str: ...

def fetch_response() -> Response: ...
```

Note that returning instances of private classes is not recommended. It is usually preferable to make such classes public.

`typing.`**`Any`**
Special type indicating an unconstrained type.

- Every type is compatible with *`Any`*.
- *`Any`* is compatible with every type.

`typing.`**`NoReturn`**
Tipo especial indicando que uma função nunca retorna. Por exemplo:

```
from typing import NoReturn

def stop() -> NoReturn:
    raise RuntimeError('no way')
```

Novo na versão 3.5.4.

Novo na versão 3.6.2.

`typing.`**`Union`**
Union type; `Union[X, Y]` means either X or Y.

To define a union, use e.g. `Union[int, str]`. Details:

- The arguments must be types and there must be at least one.
- Unions of unions are flattened, e.g.:

```
Union[Union[int, str], float] == Union[int, str, float]
```

- Unions of a single argument vanish, e.g.:

```
Union[int] == int  # The constructor actually returns int
```

- Redundant arguments are skipped, e.g.:

```
Union[int, str, int] == Union[int, str]
```

- When comparing unions, the argument order is ignored, e.g.:

```
Union[int, str] == Union[str, int]
```

- You cannot subclass or instantiate a union.
- You cannot write `Union[X][Y]`.
- You can use `Optional[X]` as a shorthand for `Union[X, None]`.

Alterado na versão 3.7: Don't remove explicit subclasses from unions at runtime.

typing.**Optional**

Tipo opcional.

`Optional[X]` is equivalent to `Union[X, None]`.

Note that this is not the same concept as an optional argument, which is one that has a default. An optional argument with a default does not require the `Optional` qualifier on its type annotation just because it is optional. For example:

```
def foo(arg: int = 0) -> None:
    ...
```

On the other hand, if an explicit value of `None` is allowed, the use of `Optional` is appropriate, whether the argument is optional or not. For example:

```
def foo(arg: Optional[int] = None) -> None:
    ...
```

typing.**Tuple**

Tuple type; `Tuple[X, Y]` is the type of a tuple of two items with the first item of type X and the second of type Y. The type of the empty tuple can be written as `Tuple[()]`.

Example: `Tuple[T1, T2]` is a tuple of two elements corresponding to type variables T1 and T2. `Tuple[int, float, str]` is a tuple of an int, a float and a string.

To specify a variable-length tuple of homogeneous type, use literal ellipsis, e.g. `Tuple[int, ...]`. A plain *Tuple* is equivalent to `Tuple[Any, ...]`, and in turn to *tuple*.

typing.**Callable**

Callable type; `Callable[[int], str]` is a function of (int) -> str.

The subscription syntax must always be used with exactly two values: the argument list and the return type. The argument list must be a list of types or an ellipsis; the return type must be a single type.

There is no syntax to indicate optional or keyword arguments; such function types are rarely used as callback types. `Callable[..., ReturnType]` (literal ellipsis) can be used to type hint a callable taking any number of arguments and returning `ReturnType`. A plain *Callable* is equivalent to `Callable[..., Any]`, and in turn to *collections.abc.Callable*.

typing.**ClassVar**
Special type construct to mark class variables.

As introduced in **PEP 526**, a variable annotation wrapped in ClassVar indicates that a given attribute is intended to be used as a class variable and should not be set on instances of that class. Usage:

```
class Starship:
    stats: ClassVar[Dict[str, int]] = {} # class variable
    damage: int = 10                     # instance variable
```

ClassVar accepts only types and cannot be further subscribed.

ClassVar is not a class itself, and should not be used with *isinstance()* or *issubclass()*. *ClassVar* does not change Python runtime behavior, but it can be used by third-party type checkers. For example, a type checker might flag the following code as an error:

```
enterprise_d = Starship(3000)
enterprise_d.stats = {} # Error, setting class variable on instance
Starship.stats = {}     # This is OK
```

Novo na versão 3.5.3.

typing.**AnyStr**
AnyStr is a type variable defined as AnyStr = TypeVar('AnyStr', str, bytes).

It is meant to be used for functions that may accept any kind of string without allowing different kinds of strings to mix. For example:

```
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat(u"foo", u"bar")  # Ok, output has type 'unicode'
concat(b"foo", b"bar")  # Ok, output has type 'bytes'
concat(u"foo", b"bar")  # Error, cannot mix unicode and bytes
```

typing.**TYPE_CHECKING**
A special constant that is assumed to be True by 3rd party static type checkers. It is False at runtime. Usage:

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: 'expensive_mod.SomeType') -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

Note that the first type annotation must be enclosed in quotes, making it a "forward reference", to hide the expensive_mod reference from the interpreter runtime. Type annotations for local variables are not evaluated, so the second annotation does not need to be enclosed in quotes.

Novo na versão 3.5.2.

27.2 pydoc — Gerador de documentação e sistema de ajuda online

Código Fonte: Lib/pydoc.py

O módulo *pydoc* gera automaticamente a documentação dos módulos Python. A documentação pode ser apresentada como páginas de texto no console, servida em um navegador web ou salva em arquivos HTML.

Para módulos, classes, funções e métodos, a documentação exibida é derivada da docstring (ou seja, o atributo `__doc__`) do objeto, e recursivamente de seus membros documentáveis. Se não houver docstring, *pydoc* tenta obter uma descrição do bloco de linhas de comentário logo acima da definição da classe, função ou método no arquivo fonte, ou no topo do módulo (consulte *inspect.getcomments()*).

A função embutida *help()* invoca o sistema de ajuda online no interpretador interativo, que usa *pydoc* para gerar sua documentação como texto no console. A mesma documentação de texto também pode ser vista de fora do interpretador Python executando **pydoc** como um script no prompt de comando do sistema operacional. Por exemplo, executar

```
pydoc sys
```

em um prompt de console exibirá a documentação do módulo *sys*, em um estilo semelhante às páginas de manual mostradas pelo comando Unix **man**. O argumento para **pydoc** pode ser o nome de uma função, módulo ou pacote, ou uma referência pontilhada a uma classe, método ou função dentro de um módulo ou módulo em um pacote. Se o argumento para **pydoc** parecer um caminho (ou seja, ele contém o separador de caminho para o seu sistema operacional, como uma barra no Unix) e se refere a um arquivo fonte Python existente, então a documentação é produzida para esse arquivo.

Nota: Para encontrar objetos e sua documentação, *pydoc* importa os módulos a serem documentados. Portanto, qualquer código no nível do módulo será executado nessa ocasião. Use uma proteção `if __name__ == '__main__':` para executar código apenas quando um arquivo é chamado como um script e não apenas importado.

Ao imprimir a saída para o console, **pydoc** tenta paginar a saída para facilitar a leitura. Se a variável de ambiente `PAGER` estiver definida, **pydoc** usará seu valor como um programa de paginação.

Especificar um sinalizador `-w` antes do argumento fará com que a documentação HTML seja escrita em um arquivo no diretório atual, ao invés de exibir texto no console.

Especificar um sinalizador `-k` antes do argumento irá pesquisar as linhas de sinopse de todos os módulos disponíveis para a palavra reservada fornecida como o argumento, novamente de uma maneira semelhante ao comando Unix **man**. A linha de sinopse de um módulo é a primeira linha de sua string de documentação.

Você também pode usar **pydoc** para iniciar um servidor HTTP na máquina local que servirá a documentação para os navegadores web visitantes. **pydoc -p 1234** irá iniciar um servidor HTTP na porta 1234, permitindo que você navegue pela documentação em `http://localhost:1234/` em seu navegador preferido. Especificar `0` como o número da porta irá selecionar uma porta não utilizada arbitrária.

pydoc -n <hostname> irá iniciar o servidor ouvindo no nome de host fornecido. Por padrão, o nome de host é "localhost", mas se você deseja que o servidor seja acessado por outras máquinas, você pode alterar o nome de host ao qual o servidor responde. Durante o desenvolvimento, isso é especialmente útil se você deseja executar o pydoc de dentro de um contêiner.

pydoc -b irá iniciar o servidor e, adicionalmente, abrir um navegador da web para uma página de índice do módulo. Cada página exibida tem uma barra de navegação na parte superior onde você pode escolher *Get* para obter ajuda em um item individual, *Search* para pesquisar todos os módulos com uma palavra reservada em sua linha de sinopse e ir para as páginas de índice do módulo em *Module index*, tópicos em *Topics* e palavras reservadas em *Keywords*.

Quando **pydoc** gera documentação, ele usa o ambiente atual e o caminho para localizar os módulos. Assim, invocar **pydoc spam** documenta precisamente a versão do módulo que você obteria se iniciasse o interpretador Python e digitasse `import spam`.

Os documentos do módulo para os módulos principais são assumidos para residir em `https://docs.python.org/X.Y/library/` onde `X` e `Y` são os números de versão principal e secundária do interpretador Python. Isso pode ser substituído definindo a variável de ambiente `PYTHONDOCS` para uma URL diferente ou para um diretório local contendo as páginas do Manual de Referência da Biblioteca.

Alterado na versão 3.2: Adicionada a opção `-b`.

Alterado na versão 3.3: A opção de linha de comando `-g` foi removida.

Alterado na versão 3.4: *pydoc* agora usa *inspect.signature()* em vez de *inspect.getfullargspec()* para extrair informações de assinatura de chamáveis.

Alterado na versão 3.7: Adicionada a opção `-n`.

27.3 doctest — Teste exemplos interativos de Python

Source code: Lib/doctest.py

O módulo *doctest* busca partes de texto que se parecem com sessões interativas do Python e, em seguida, executa essas sessões para verificar se elas funcionam exatamente como mostrado. Existem várias maneiras comuns de usar o doctest:

- Para verificar se as docstrings de um módulo estão atualizadas, verificando que todos os exemplos interativos ainda funcionam conforme documentado.
- Para executar testes de regressão, verificando que os exemplos interativos de um arquivo de teste ou um objeto de teste funcionam como esperado.
- Para escrever a documentação do tutorial para um pacote, ilustrado de forma liberal com exemplos de entrada e saída. Dependendo se os exemplos ou o texto expositivo são enfatizados, isso tem o sabor do "teste ilustrado" ou "documentação executável".

Aqui temos um pequeno exemplo, porém, completo:

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
```

(continua na próxima página)

(continuação da página anterior)

```
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Se executares diretamente `example.py` desde a linha de comando, *doctest* a mágica funcionará:

```
$ python example.py
$
```

Observe que nada foi impresso na saída padrão! Isso é normal, e isso significa que todos os exemplos funcionaram. Passe `-v` para o script e *doctest* imprimirá um registro detalhado do que está sendo testando imprimindo ainda um resumo no final:

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

E assim por diante, eventualmente terminando com:

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items passed all tests:
   1 tests in __main__
   8 tests in __main__.factorial
9 tests in 2 items.
9 passed and 0 failed.
Test passed.
$
```

Isso é tudo o que precisas saber para começar a fazer uso produtivo do módulo *doctest*! Pule! As seções a seguir fornecem detalhes completos. Observe que há muitos exemplos de documentos no conjunto de teste padrão Python e bibliotecas. Exemplos especialmente úteis podem ser encontrados no arquivo de teste padrão `Lib/test/test_doctest.py`.

27.3.1 Uso simples: verificando exemplos em Docstrings

A maneira mais simples de começar a usar o doctest (mas não necessariamente a maneira como você continuará fazendo isso) é encerrar cada módulo `M` com:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

doctest e então examine a docstrings no módulo `M`.

Executar o módulo como um Script faz com que os exemplos nas docstrings sejam executados e verificados.

```
python M.py
```

Isso não exibirá nada, a menos que um exemplo falhe, caso em que o(s) exemplo(s) falhando(s) e a(s) causa(s) da(s) falha(s) são impressas em stdout e a linha final de saída será `***Test Failed*** N failures.`, onde *N* é o número de exemplos que falharam.

Ao invés disso, execute agora com a opção `-v`:

```
python M.py -v
```

and a detailed report of all examples tried is printed to standard output, along with assorted summaries at the end.

You can force verbose mode by passing `verbose=True` to *testmod()*, or prohibit it by passing `verbose=False`. In either of those cases, `sys.argv` is not examined by *testmod()* (so passing `-v` or not has no effect).

There is also a command line shortcut for running *testmod()*. You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the module name(s) on the command line:

```
python -m doctest -v example.py
```

This will import `example.py` as a standalone module and run *testmod()* on it. Note that this may not work correctly if the file is part of a package and imports other submodules from that package.

For more information on *testmod()*, see section *Basic API*.

27.3.2 Utilização comum: Verificando exemplos em um arquivo de texto

Another simple application of doctest is testing interactive examples in a text file. This can be done with the *testfile()* function:

```
import doctest
doctest.testfile("example.txt")
```

That short script executes and verifies any interactive Python examples contained in the file `example.txt`. The file content is treated as if it were a single giant docstring; the file doesn't need to contain a Python program! For example, perhaps `example.txt` contains this:

```
The ``example`` module
======================

Using ``factorial``
-------------------

This is an example text file in reStructuredText format.  First import
``factorial`` from the ``example`` module:

    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

Running `doctest.testfile("example.txt")` then finds the error in this documentation:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

As with *testmod()*, *testfile()* won't display anything unless an example fails. If an example does fail, then the failing example(s) and the cause(s) of the failure(s) are printed to stdout, using the same format as *testmod()*.

By default, *testfile()* looks for files in the calling module's directory. See section *Basic API* for a description of the optional arguments that can be used to tell it to look for files in other locations.

Like *testmod()*, *testfile()*'s verbosity can be set with the `-v` command-line switch or with the optional keyword argument *verbose*.

There is also a command line shortcut for running *testfile()*. You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the file name(s) on the command line:

```
python -m doctest -v example.txt
```

Because the file name does not end with `.py`, *doctest* infers that it must be run with *testfile()*, not *testmod()*.

Para maiores informações em *testfile()*, veja a seção *Basic API*.

27.3.3 Como funciona

This section examines in detail how doctest works: which docstrings it looks at, how it finds interactive examples, what execution context it uses, how it handles exceptions, and how option flags can be used to control its behavior. This is the information that you need to know to write doctest examples; for information about actually running doctest on these examples, see the following sections.

Which Docstrings Are Examined?

The module docstring, and all function, class and method docstrings are searched. Objects imported into the module are not searched.

In addition, if `M.__test__` exists and "is true", it must be a dict, and each entry maps a (string) name to a function object, class object, or string. Function and class object docstrings found from `M.__test__` are searched, and strings are treated as if they were docstrings. In output, a key `K` in `M.__test__` appears with name

```
<name of M>.__test__.K
```

Any classes found are recursively searched similarly, to test docstrings in their contained methods and nested classes.

CPython implementation detail: Prior to version 3.4, extension modules written in C were not fully searched by doctest.

How are Docstring Examples Recognized?

In most cases a copy-and-paste of an interactive console session works fine, but doctest isn't trying to do an exact emulation of any specific Python shell.

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
...
no
NO
NO!!!
>>>
```

Any expected output must immediately follow the final `'>>> '` or `'... '` line containing the code, and the expected output (if any) extends to the next `'>>> '` or all-whitespace line.

The fine print:

- Expected output cannot contain an all-whitespace line, since such a line is taken to signal the end of expected output. If expected output does contain a blank line, put `<BLANKLINE>` in your doctest example each place a blank line is expected.
- All hard tab characters are expanded to spaces, using 8-column tab stops. Tabs in output generated by the tested code are not modified. Because any hard tabs in the sample output *are* expanded, this means that if the code output includes hard tabs, the only way the doctest can pass is if the *NORMALIZE_WHITESPACE* option or *directive* is in effect. Alternatively, the test can be rewritten to capture the output and compare it to an expected value as part of the test. This handling of tabs in the source was arrived at through trial and error, and has proven to be the least

error prone way of handling them. It is possible to use a different algorithm for handling tabs by writing a custom *DocTestParser* class.

- Output to stdout is captured, but not output to stderr (exception tracebacks are captured via a different means).
- If you continue a line via backslashing in an interactive session, or for any other reason use a backslash, you should use a raw docstring, which will preserve your backslashes exactly as you type them:

```
>>> def f(x):
...     r'''Backslashes in a raw docstring: m\n'''
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

Otherwise, the backslash will be interpreted as part of the string. For example, the `\n` above would be interpreted as a newline character. Alternatively, you can double each backslash in the doctest version (and not use a raw string):

```
>>> def f(x):
...     '''Backslashes in a raw docstring: m\\n'''
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

- The starting column doesn't matter:

```
>>> assert "Easy!"
      >>> import math
          >>> math.floor(1.9)
          1
```

and as many leading whitespace characters are stripped from the expected output as appeared in the initial `'>>> '` line that started the example.

What's the Execution Context?

By default, each time *doctest* finds a docstring to test, it uses a *shallow copy* of `M`'s globals, so that running tests doesn't change the module's real globals, and so that one test in `M` can't leave behind crumbs that accidentally allow another test to work. This means examples can freely use any names defined at top-level in `M`, and names defined earlier in the docstring being run. Examples cannot see names defined in other docstrings.

You can force use of your own dict as the execution context by passing `globs=your_dict` to *testmod()* or *testfile()* instead.

What About Exceptions?

No problem, provided that the traceback is the only output produced by the example: just paste in the traceback.[1] Since tracebacks contain details that are likely to change rapidly (for example, exact file paths and line numbers), this is one case where doctest works hard to be flexible in what it accepts.

Simple example:

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

[1] Examples containing both expected output and an exception are not supported. Trying to guess where one ends and the other begins is too error-prone, and that also makes for a confusing test.

That doctest succeeds if *ValueError* is raised, with the `list.remove(x): x not in list` detail as shown.

The expected output for an exception must start with a traceback header, which may be either of the following two lines, indented the same as the first line of the example:

```
Traceback (most recent call last):
Traceback (innermost last):
```

The traceback header is followed by an optional traceback stack, whose contents are ignored by doctest. The traceback stack is typically omitted, or copied verbatim from an interactive session.

The traceback stack is followed by the most interesting part: the line(s) containing the exception type and detail. This is usually the last line of a traceback, but can extend across multiple lines if the exception has a multi-line detail:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

The last three lines (starting with *ValueError*) are compared against the exception's type and detail, and the rest are ignored.

Best practice is to omit the traceback stack, unless it adds significant documentation value to the example. So the last example is probably better as:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
    ...
ValueError: multi
    line
detail
```

Note that tracebacks are treated very specially. In particular, in the rewritten example, the use of `...` is independent of doctest's *ELLIPSIS* option. The ellipsis in that example could be left out, or could just as well be three (or three hundred) commas or digits, or an indented transcript of a Monty Python skit.

Some details you should read once, but won't need to remember:

- Doctest can't guess whether your expected output came from an exception traceback or from ordinary printing. So, e.g., an example that expects `ValueError: 42 is prime` will pass whether *ValueError* is actually raised or if the example merely prints that traceback text. In practice, ordinary output rarely begins with a traceback header line, so this doesn't create real problems.
- Each line of the traceback stack (if present) must be indented further than the first line of the example, *or* start with a non-alphanumeric character. The first line following the traceback header indented the same and starting with an alphanumeric is taken to be the start of the exception detail. Of course this does the right thing for genuine tracebacks.
- When the *IGNORE_EXCEPTION_DETAIL* doctest option is specified, everything following the leftmost colon and any module information in the exception name is ignored.
- The interactive shell omits the traceback header line for some *SyntaxError*s. But doctest uses the traceback header line to distinguish exceptions from non-exceptions. So in the rare case where you need to test a *SyntaxError* that omits the traceback header, you will need to manually add the traceback header line to your test example.
- For some *SyntaxError*s, Python displays the character position of the syntax error, using a ^ marker:

```
>>> 1 1
  File "<stdin>", line 1
    1 1
      ^
SyntaxError: invalid syntax
```

Since the lines showing the position of the error come before the exception type and detail, they are not checked by doctest. For example, the following test would pass, even though it puts the ^ marker in the wrong location:

```
>>> 1 1
  File "<stdin>", line 1
    1 1
    ^
SyntaxError: invalid syntax
```

Flags opcionais

A number of option flags control various aspects of doctest's behavior. Symbolic names for the flags are supplied as module constants, which can be bitwise ORed together and passed to various functions. The names can also be used in *doctest directives*, and may be passed to the doctest command line interface via the `-o` option.

Novo na versão 3.4: The `-o` command line option.

The first group of options define test semantics, controlling aspects of how doctest decides whether actual output matches an example's expected output:

`doctest.`**`DONT_ACCEPT_TRUE_FOR_1`**
: By default, if an expected output block contains just `1`, an actual output block containing just `1` or just `True` is considered to be a match, and similarly for `0` versus `False`. When *`DONT_ACCEPT_TRUE_FOR_1`* is specified, neither substitution is allowed. The default behavior caters to that Python changed the return type of many functions from integer to boolean; doctests expecting "little integer" output still work in these cases. This option will probably go away, but not for several years.

`doctest.`**`DONT_ACCEPT_BLANKLINE`**
: By default, if an expected output block contains a line containing only the string `<BLANKLINE>`, then that line will match a blank line in the actual output. Because a genuinely blank line delimits the expected output, this is the only way to communicate that a blank line is expected. When *`DONT_ACCEPT_BLANKLINE`* is specified, this substitution is not allowed.

`doctest.`**`NORMALIZE_WHITESPACE`**
: When specified, all sequences of whitespace (blanks and newlines) are treated as equal. Any sequence of whitespace within the expected output will match any sequence of whitespace within the actual output. By default, whitespace must match exactly. *`NORMALIZE_WHITESPACE`* is especially useful when a line of expected output is very long, and you want to wrap it across multiple lines in your source.

`doctest.`**`ELLIPSIS`**
: When specified, an ellipsis marker (`...`) in the expected output can match any substring in the actual output. This includes substrings that span line boundaries, and empty substrings, so it's best to keep usage of this simple. Complicated uses can lead to the same kinds of "oops, it matched too much!" surprises that `.*` is prone to in regular expressions.

`doctest.`**`IGNORE_EXCEPTION_DETAIL`**
: When specified, an example that expects an exception passes if an exception of the expected type is raised, even if the exception detail does not match. For example, an example expecting `ValueError: 42` will pass if the actual exception raised is `ValueError: 3*14`, but will fail, e.g., if *`TypeError`* is raised.

It will also ignore the module name used in Python 3 doctest reports. Hence both of these variations will work with the flag specified, regardless of whether the test is run under Python 2.7 or Python 3.2 (or later versions):

```
>>> raise CustomError('message')
Traceback (most recent call last):
CustomError: message

>>> raise CustomError('message')
Traceback (most recent call last):
my_module.CustomError: message
```

Note that *ELLIPSIS* can also be used to ignore the details of the exception message, but such a test may still fail based on whether or not the module details are printed as part of the exception name. Using *IGNORE_EXCEPTION_DETAIL* and the details from Python 2.3 is also the only clear way to write a doctest that doesn't care about the exception detail yet continues to pass under Python 2.3 or earlier (those releases do not support *doctest directives* and ignore them as irrelevant comments). For example:

```
>>> (1, 2)[3] = 'moo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object doesn't support item assignment
```

passes under Python 2.3 and later Python versions with the flag specified, even though the detail changed in Python 2.4 to say "does not" instead of "doesn't".

Alterado na versão 3.2: *IGNORE_EXCEPTION_DETAIL* now also ignores any information relating to the module containing the exception under test.

doctest.**SKIP**
: When specified, do not run the example at all. This can be useful in contexts where doctest examples serve as both documentation and test cases, and an example should be included for documentation purposes, but should not be checked. E.g., the example's output might be random; or the example might depend on resources which would be unavailable to the test driver.

  The SKIP flag can also be used for temporarily "commenting out" examples.

doctest.**COMPARISON_FLAGS**
: A bitmask or'ing together all the comparison flags above.

The second group of options controls how test failures are reported:

doctest.**REPORT_UDIFF**
: When specified, failures that involve multi-line expected and actual outputs are displayed using a unified diff.

doctest.**REPORT_CDIFF**
: When specified, failures that involve multi-line expected and actual outputs will be displayed using a context diff.

doctest.**REPORT_NDIFF**
: When specified, differences are computed by `difflib.Differ`, using the same algorithm as the popular `ndiff.py` utility. This is the only method that marks differences within lines as well as across lines. For example, if a line of expected output contains digit `1` where actual output contains letter `l`, a line is inserted with a caret marking the mismatching column positions.

doctest.**REPORT_ONLY_FIRST_FAILURE**
: When specified, display the first failing example in each doctest, but suppress output for all remaining examples. This will prevent doctest from reporting correct examples that break because of earlier failures; but it might also hide incorrect examples that fail independently of the first failure. When *REPORT_ONLY_FIRST_FAILURE* is specified, the remaining examples are still run, and still count towards the total number of failures reported; only the output is suppressed.

doctest.**FAIL_FAST**
: When specified, exit after the first failing example and don't attempt to run the remaining examples. Thus, the number of failures reported will be at most 1. This flag may be useful during debugging, since examples after the first failure won't even produce debugging output.

  The doctest command line accepts the option `-f` as a shorthand for `-o FAIL_FAST`.

  Novo na versão 3.4.

doctest.**REPORTING_FLAGS**
: A bitmask or'ing together all the reporting flags above.

There is also a way to register new option flag names, though this isn't useful unless you intend to extend *doctest* internals via subclassing:

doctest.**register_optionflag**(*name*)
: Create a new option flag with a given name, and return the new flag's integer value. *register_optionflag()* can be used when subclassing *OutputChecker* or *DocTestRunner* to create new options that are supported by your subclasses. *register_optionflag()* should always be called using the following idiom:

```
MY_FLAG = register_optionflag('MY_FLAG')
```

Directives

Doctest directives may be used to modify the *option flags* for an individual example. Doctest directives are special Python comments following an example's source code:

```
directive             ::=   "#" "doctest:" directive_options
directive_options     ::=   directive_option ("," directive_option)\*
directive_option      ::=   on_or_off directive_option_name
on_or_off             ::=   "+" \| "-"
directive_option_name ::=   "DONT_ACCEPT_BLANKLINE" \| "NORMALIZE_WHITESPACE" \| ...
```

Whitespace is not allowed between the + or − and the directive option name. The directive option name can be any of the option flag names explained above.

An example's doctest directives modify doctest's behavior for that single example. Use + to enable the named behavior, or − to disable it.

For example, this test passes:

```
>>> print(list(range(20)))
[0,   1,  2,  3,  4,  5,  6,  7,  8,  9,
10,  11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Without the directive it would fail, both because the actual output doesn't have two blanks before the single-digit list elements, and because the actual output is on a single line. This test also passes, and also requires a directive to do so:

```
>>> print(list(range(20)))
[0, 1, ..., 18, 19]
```

Multiple directives can be used on a single physical line, separated by commas:

```
>>> print(list(range(20)))
[0,    1, ...,   18,    19]
```

If multiple directive comments are used for a single example, then they are combined:

```
>>> print(list(range(20)))
...
[0,    1, ...,   18,    19]
```

As the previous example shows, you can add `...` lines to your example containing only directives. This can be useful when an example is too long for a directive to comfortably fit on the same line:

```
>>> print(list(range(5)) + list(range(10, 20)) + list(range(30, 40)))
...
[0, ..., 4, 10, ..., 19, 30, ..., 39]
```

Note that since all options are disabled by default, and directives apply only to the example they appear in, enabling options (via + in a directive) is usually the only meaningful choice. However, option flags can also be passed to functions that run doctests, establishing different defaults. In such cases, disabling an option via − in a directive can be useful.

Avisos

doctest is serious about requiring exact matches in expected output. If even a single character doesn't match, the test fails. This will probably surprise you a few times, as you learn exactly what Python does and doesn't guarantee about output. For example, when printing a set, Python doesn't guarantee that the element is printed in any particular order, so a test like

```
>>> foo()
{"Hermione", "Harry"}
```

is vulnerable! One workaround is to do

```
>>> foo() == {"Hermione", "Harry"}
True
```

instead. Another is to do

```
>>> d = sorted(foo())
>>> d
['Harry', 'Hermione']
```

Nota: Before Python 3.6, when printing a dict, Python did not guarantee that the key-value pairs was printed in any particular order.

There are others, but you get the idea.

Another bad idea is to print things that embed an object address, like

```
>>> id(1.0) # certain to fail some of the time
7948648
>>> class C: pass
>>> C()   # the default repr() for instances embeds an address
<__main__.C instance at 0x00AC18F0>
```

The *ELLIPSIS* directive gives a nice approach for the last example:

```
>>> C()
<__main__.C instance at 0x...>
```

Floating-point numbers are also subject to small output variations across platforms, because Python defers to the platform C library for float formatting, and C libraries vary widely in quality here.

```
>>> 1./7  # risky
0.14285714285714285
>>> print(1./7) # safer
0.142857142857
>>> print(round(1./7, 6)) # much safer
0.142857
```

Numbers of the form `I/2.**J` are safe across all platforms, and I often contrive doctest examples to produce numbers of that form:

```
>>> 3./4  # utterly safe
0.75
```

Simple fractions are also easier for people to understand, and that makes for better documentation.

27.3.4 Basic API

The functions `testmod()` and `testfile()` provide a simple interface to doctest that should be sufficient for most basic uses. For a less formal introduction to these two functions, see sections *Uso simples: verificando exemplos em Docstrings* and *Utilização comum: Verificando exemplos em um arquivo de texto*.

doctest.**testfile**(*filename*, *module_relative=True*, *name=None*, *package=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *parser=DocTestParser()*, *encoding=None*)

All arguments except *filename* are optional, and should be specified in keyword form.

Test examples in the file named *filename*. Return `(failure_count, test_count)`.

Optional argument *module_relative* specifies how the filename should be interpreted:

- If *module_relative* is `True` (the default), then *filename* specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the *package* argument is specified, then it is relative to that package. To ensure OS-independence, *filename* should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If *module_relative* is `False`, then *filename* specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument *name* gives the name of the test; by default, or if `None`, `os.path.basename(filename)` is used.

Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for a module-relative filename. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is `False`.

Optional argument *globs* gives a dict to be used as the globals when executing examples. A new shallow copy of this dict is created for the doctest, so its examples start with a clean slate. By default, or if `None`, a new empty dict is used.

Optional argument *extraglobs* gives a dict merged into the globals used to execute examples. This works like `dict.update()`: if *globs* and *extraglobs* have a common key, the associated value in *extraglobs* appears in the combined dict. By default, or if `None`, no extra globals are used. This is an advanced feature that allows parameterization of doctests. For example, a doctest can be written for a base class, using a generic name for the class, then reused to test any number of subclasses by passing an *extraglobs* dict mapping the generic name to the subclass to be tested.

Optional argument *verbose* prints lots of stuff if true, and prints only failures if false; by default, or if `None`, it's true if and only if `'-v'` is in `sys.argv`.

Optional argument *report* prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else the summary is very brief (in fact, empty if all tests passed).

Optional argument *optionflags* (default value 0) takes the bitwise OR of option flags. See section *Flags opcionais*.

Optional argument *raise_on_error* defaults to false. If true, an exception is raised upon the first failure or unexpected exception in an example. This allows failures to be post-mortem debugged. Default behavior is to continue running examples.

Optional argument *parser* specifies a `DocTestParser` (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

`doctest.`**`testmod`**(*m=None*, *name=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *exclude_empty=False*)

All arguments are optional, and all except for *m* should be specified in keyword form.

Test examples in docstrings in functions and classes reachable from module *m* (or module `__main__` if *m* is not supplied or is `None`), starting with `m.__doc__`.

Also test examples reachable from dict `m.__test__`, if it exists and is not `None`. `m.__test__` maps names (strings) to functions, classes and strings; function and class docstrings are searched for examples; strings are searched directly, as if they were docstrings.

Only docstrings attached to objects belonging to module *m* are searched.

Return `(failure_count, test_count)`.

Optional argument *name* gives the name of the module; by default, or if `None`, `m.__name__` is used.

Optional argument *exclude_empty* defaults to false. If true, objects for which no doctests are found are excluded from consideration. The default is a backward compatibility hack, so that code still using `doctest.master.summarize()` in conjunction with `testmod()` continues to get output for objects with no tests. The *exclude_empty* argument to the newer `DocTestFinder` constructor defaults to true.

Optional arguments *extraglobs*, *verbose*, *report*, *optionflags*, *raise_on_error*, and *globs* are the same as for function `testfile()` above, except that *globs* defaults to `m.__dict__`.

`doctest.`**`run_docstring_examples`**(*f*, *globs*, *verbose=False*, *name="NoName"*, *compileflags=None*, *optionflags=0*)

Test examples associated with object *f*; for example, *f* may be a string, a module, a function, or a class object.

A shallow copy of dictionary argument *globs* is used for the execution context.

Optional argument *name* is used in failure messages, and defaults to `"NoName"`.

If optional argument *verbose* is true, output is generated even if there are no failures. By default, output is generated only in case of an example failure.

Optional argument *compileflags* gives the set of flags that should be used by the Python compiler when running the examples. By default, or if `None`, flags are deduced corresponding to the set of future features found in *globs*.

Optional argument *optionflags* works as for function `testfile()` above.

27.3.5 API do Unittest

As your collection of doctest'ed modules grows, you'll want a way to run all their doctests systematically. *doctest* provides two functions that can be used to create *unittest* test suites from modules and text files containing doctests. To integrate with *unittest* test discovery, include a `load_tests()` function in your test module:

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

There are two main functions for creating *unittest.TestSuite* instances from text files and modules with doctests:

doctest.**DocFileSuite**(**paths*, *module_relative=True*, *package=None*, *setUp=None*, *tearDown=None*, *globs=None*, *optionflags=0*, *parser=DocTestParser()*, *encoding=None*)

Convert doctest tests from one or more text files to a *unittest.TestSuite*.

The returned *unittest.TestSuite* is to be run by the unittest framework and runs the interactive examples in each file. If an example in any file fails, then the synthesized unit test fails, and a `failureException` exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Pass one or more paths (as strings) to text files to be examined.

Options may be provided as keyword arguments:

Optional argument *module_relative* specifies how the filenames in *paths* should be interpreted:

- If *module_relative* is `True` (the default), then each filename in *paths* specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the *package* argument is specified, then it is relative to that package. To ensure OS-independence, each filename should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If *module_relative* is `False`, then each filename in *paths* specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for module-relative filenames in *paths*. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is `False`.

Optional argument *setUp* specifies a set-up function for the test suite. This is called before running the tests in each file. The *setUp* function will be passed a *DocTest* object. The setUp function can access the test globals as the *globs* attribute of the test passed.

Optional argument *tearDown* specifies a tear-down function for the test suite. This is called after running the tests in each file. The *tearDown* function will be passed a *DocTest* object. The setUp function can access the test globals as the *globs* attribute of the test passed.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is a new empty dictionary.

Optional argument *optionflags* specifies the default doctest options for the tests, created by or-ing together individual option flags. See section *Flags opcionais*. See function *set_unittest_reportflags()* below for a better way to set reporting options.

Optional argument *parser* specifies a *DocTestParser* (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

The global `__file__` is added to the globals provided to doctests loaded from a text file using *DocFileSuite()*.

doctest.**DocTestSuite**(*module=None*, *globs=None*, *extraglobs=None*, *test_finder=None*, *setUp=None*, *tearDown=None*, *checker=None*)

Convert doctest tests for a module to a *unittest.TestSuite*.

The returned *unittest.TestSuite* is to be run by the unittest framework and runs each doctest in the module. If any of the doctests fail, then the synthesized unit test fails, and a `failureException` exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Optional argument *module* provides the module to be tested. It can be a module object or a (possibly dotted) module name. If not specified, the module calling this function is used.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is a new empty dictionary.

Optional argument *extraglobs* specifies an extra set of global variables, which is merged into *globs*. By default, no extra globals are used.

Optional argument *test_finder* is the *DocTestFinder* object (or a drop-in replacement) that is used to extract doctests from the module.

Optional arguments *setUp*, *tearDown*, and *optionflags* are the same as for function *DocFileSuite()* above.

This function uses the same search technique as *testmod()*.

Alterado na versão 3.5: *DocTestSuite()* returns an empty *unittest.TestSuite* if *module* contains no docstrings instead of raising *ValueError*.

Under the covers, *DocTestSuite()* creates a *unittest.TestSuite* out of `doctest.DocTestCase` instances, and `DocTestCase` is a subclass of *unittest.TestCase*. `DocTestCase` isn't documented here (it's an internal detail), but studying its code can answer questions about the exact details of *unittest* integration.

Similarly, *DocFileSuite()* creates a *unittest.TestSuite* out of `doctest.DocFileCase` instances, and `DocFileCase` is a subclass of `DocTestCase`.

So both ways of creating a *unittest.TestSuite* run instances of `DocTestCase`. This is important for a subtle reason: when you run *doctest* functions yourself, you can control the *doctest* options in use directly, by passing option flags to *doctest* functions. However, if you're writing a *unittest* framework, *unittest* ultimately controls when and how tests get run. The framework author typically wants to control *doctest* reporting options (perhaps, e.g., specified by command line options), but there's no way to pass options through *unittest* to *doctest* test runners.

For this reason, *doctest* also supports a notion of *doctest* reporting flags specific to *unittest* support, via this function:

doctest.**set_unittest_reportflags**(*flags*)

Set the *doctest* reporting flags to use.

Argument *flags* takes the bitwise OR of option flags. See section *Flags opcionais*. Only "reporting flags" can be used.

This is a module-global setting, and affects all future doctests run by module *unittest*: the `runTest()` method of `DocTestCase` looks at the option flags specified for the test case when the `DocTestCase` instance was constructed. If no reporting flags were specified (which is the typical and expected case), *doctest*'s *unittest* reporting flags are bitwise ORed into the option flags, and the option flags so augmented are passed to the *DocTestRunner* instance created to run the doctest. If any reporting flags were specified when the `DocTestCase` instance was constructed, *doctest*'s *unittest* reporting flags are ignored.

The value of the *unittest* reporting flags in effect before the function was called is returned by the function.

27.3.6 Advanced API

The basic API is a simple wrapper that's intended to make doctest easy to use. It is fairly flexible, and should meet most users' needs; however, if you require more fine-grained control over testing, or wish to extend doctest's capabilities, then you should use the advanced API.

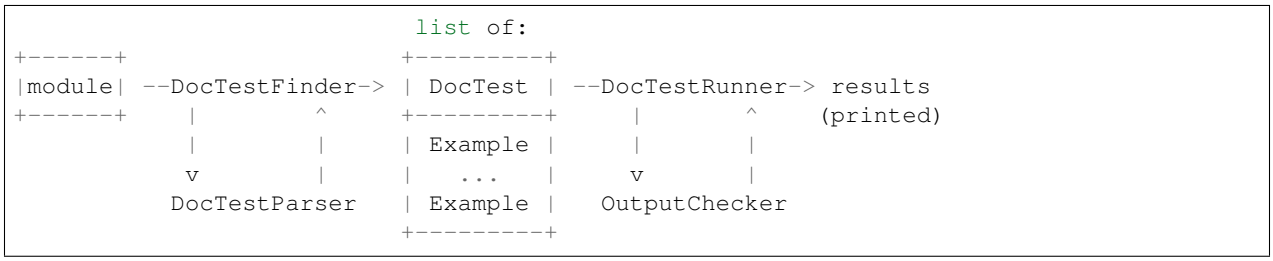
The advanced API revolves around two container classes, which are used to store the interactive examples extracted from doctest cases:

- *Example*: A single Python *statement*, paired with its expected output.
- *DocTest*: A collection of *Example*s, typically extracted from a single docstring or text file.

Additional processing classes are defined to find, parse, and run, and check doctest examples:

- *DocTestFinder*: Finds all docstrings in a given module, and uses a *DocTestParser* to create a *DocTest* from every docstring that contains interactive examples.
- *DocTestParser*: Creates a *DocTest* object from a string (such as an object's docstring).
- *DocTestRunner*: Executes the examples in a *DocTest*, and uses an *OutputChecker* to verify their output.
- *OutputChecker*: Compares the actual output from a doctest example with the expected output, and decides whether they match.

The relationships among these processing classes are summarized in the following diagram:



DocTest Objects

class doctest.**DocTest**(*examples*, *globs*, *name*, *filename*, *lineno*, *docstring*)

A collection of doctest examples that should be run in a single namespace. The constructor arguments are used to initialize the attributes of the same names.

DocTest defines the following attributes. They are initialized by the constructor, and should not be modified directly.

examples

A list of *Example* objects encoding the individual interactive Python examples that should be run by this test.

globs

The namespace (aka globals) that the examples should be run in. This is a dictionary mapping names to values. Any changes to the namespace made by the examples (such as binding new variables) will be reflected in *globs* after the test is run.

name

A string name identifying the *DocTest*. Typically, this is the name of the object or file that the test was extracted from.

filename

The name of the file that this *DocTest* was extracted from; or `None` if the filename is unknown, or if the *DocTest* was not extracted from a file.

lineno
The line number within *filename* where this *DocTest* begins, or None if the line number is unavailable. This line number is zero-based with respect to the beginning of the file.

docstring
The string that the test was extracted from, or None if the string is unavailable, or if the test was not extracted from a string.

Example Objects

class doctest.**Example**(*source*, *want*, *exc_msg=None*, *lineno=0*, *indent=0*, *options=None*)
A single interactive example, consisting of a Python statement and its expected output. The constructor arguments are used to initialize the attributes of the same names.

Example defines the following attributes. They are initialized by the constructor, and should not be modified directly.

source
A string containing the example's source code. This source code consists of a single Python statement, and always ends with a newline; the constructor adds a newline when necessary.

want
The expected output from running the example's source code (either from stdout, or a traceback in case of exception). *want* ends with a newline unless no output is expected, in which case it's an empty string. The constructor adds a newline when necessary.

exc_msg
The exception message generated by the example, if the example is expected to generate an exception; or None if it is not expected to generate an exception. This exception message is compared against the return value of *traceback.format_exception_only()*. *exc_msg* ends with a newline unless it's None. The constructor adds a newline if needed.

lineno
The line number within the string containing this example where the example begins. This line number is zero-based with respect to the beginning of the containing string.

indent
The example's indentation in the containing string, i.e., the number of space characters that precede the example's first prompt.

options
A dictionary mapping from option flags to True or False, which is used to override default options for this example. Any option flags not contained in this dictionary are left at their default value (as specified by the *DocTestRunner*'s optionflags). By default, no options are set.

DocTestFinder objects

class doctest.**DocTestFinder**(*verbose=False*, *parser=DocTestParser()*, *recurse=True*, *exclude_empty=True*)
A processing class used to extract the *DocTest*s that are relevant to a given object, from its docstring and the docstrings of its contained objects. *DocTest*s can be extracted from modules, classes, functions, methods, staticmethods, classmethods, and properties.

The optional argument *verbose* can be used to display the objects searched by the finder. It defaults to False (no output).

The optional argument *parser* specifies the *DocTestParser* object (or a drop-in replacement) that is used to extract doctests from docstrings.

If the optional argument *recurse* is false, then `DocTestFinder.find()` will only examine the given object, and not any contained objects.

If the optional argument *exclude_empty* is false, then `DocTestFinder.find()` will include tests for objects with empty docstrings.

`DocTestFinder` defines the following method:

find(*obj[, name][, module][, globs][, extraglobs]*)
Return a list of the `DocTest`s that are defined by *obj*'s docstring, or by any of its contained objects' docstrings.

The optional argument *name* specifies the object's name; this name will be used to construct names for the returned `DocTest`s. If *name* is not specified, then `obj.__name__` is used.

The optional parameter *module* is the module that contains the given object. If the module is not specified or is `None`, then the test finder will attempt to automatically determine the correct module. The object's module is used:

- As a default namespace, if *globs* is not specified.
- To prevent the DocTestFinder from extracting DocTests from objects that are imported from other modules. (Contained objects with modules other than *module* are ignored.)
- To find the name of the file containing the object.
- To help find the line number of the object within its file.

If *module* is `False`, no attempt to find the module will be made. This is obscure, of use mostly in testing doctest itself: if *module* is `False`, or is `None` but cannot be found automatically, then all objects are considered to belong to the (non-existent) module, so all contained objects will (recursively) be searched for doctests.

The globals for each `DocTest` is formed by combining *globs* and *extraglobs* (bindings in *extraglobs* override bindings in *globs*). A new shallow copy of the globals dictionary is created for each `DocTest`. If *globs* is not specified, then it defaults to the module's *__dict__*, if specified, or `{}` otherwise. If *extraglobs* is not specified, then it defaults to `{}`.

DocTestParser objects

class doctest.**DocTestParser**
A processing class used to extract interactive examples from a string, and use them to create a `DocTest` object.

`DocTestParser` defines the following methods:

get_doctest(*string*, *globs*, *name*, *filename*, *lineno*)
Extract all doctest examples from the given string, and collect them into a `DocTest` object.

globs, *name*, *filename*, and *lineno* are attributes for the new `DocTest` object. See the documentation for `DocTest` for more information.

get_examples(*string*, *name='<string>'*)
Extract all doctest examples from the given string, and return them as a list of `Example` objects. Line numbers are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

parse(*string*, *name='<string>'*)
Divide the given string into examples and intervening text, and return them as a list of alternating `Example`s and strings. Line numbers for the `Example`s are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

DocTestRunner objects

class doctest.**DocTestRunner**(*checker=None*, *verbose=None*, *optionflags=0*)
A processing class used to execute and verify the interactive examples in a *DocTest*.

The comparison between expected outputs and actual outputs is done by an *OutputChecker*. This comparison may be customized with a number of option flags; see section *Flags opcionais* for more information. If the option flags are insufficient, then the comparison may also be customized by passing a subclass of *OutputChecker* to the constructor.

The test runner's display output can be controlled in two ways. First, an output function can be passed to `TestRunner.run()`; this function will be called with strings that should be displayed. It defaults to `sys.stdout.write`. If capturing the output is not sufficient, then the display output can be also customized by subclassing DocTestRunner, and overriding the methods *report_start()*, *report_success()*, *report_unexpected_exception()*, and *report_failure()*.

The optional keyword argument *checker* specifies the *OutputChecker* object (or drop-in replacement) that should be used to compare the expected outputs to the actual outputs of doctest examples.

The optional keyword argument *verbose* controls the *DocTestRunner*'s verbosity. If *verbose* is `True`, then information is printed about each example, as it is run. If *verbose* is `False`, then only failures are printed. If *verbose* is unspecified, or `None`, then verbose output is used iff the command-line switch `-v` is used.

The optional keyword argument *optionflags* can be used to control how the test runner compares expected output to actual output, and how it displays failures. For more information, see section *Flags opcionais*.

DocTestParser defines the following methods:

report_start(*out*, *test*, *example*)
Report that the test runner is about to process the given example. This method is provided to allow subclasses of *DocTestRunner* to customize their output; it should not be called directly.

example is the example about to be processed. *test* is the test *containing example*. *out* is the output function that was passed to *DocTestRunner.run()*.

report_success(*out*, *test*, *example*, *got*)
Report that the given example ran successfully. This method is provided to allow subclasses of *DocTestRunner* to customize their output; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to *DocTestRunner.run()*.

report_failure(*out*, *test*, *example*, *got*)
Report that the given example failed. This method is provided to allow subclasses of *DocTestRunner* to customize their output; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to *DocTestRunner.run()*.

report_unexpected_exception(*out*, *test*, *example*, *exc_info*)
Report that the given example raised an unexpected exception. This method is provided to allow subclasses of *DocTestRunner* to customize their output; it should not be called directly.

example is the example about to be processed. *exc_info* is a tuple containing information about the unexpected exception (as returned by *sys.exc_info()*). *test* is the test containing *example*. *out* is the output function that was passed to *DocTestRunner.run()*.

run(*test*, *compileflags=None*, *out=None*, *clear_globs=True*)
Run the examples in *test* (a *DocTest* object), and display the results using the writer function *out*.

The examples are run in the namespace `test.globs`. If *clear_globs* is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use *clear_globs=False*.

compileflags gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to *globs*.

The output of each example is checked using the *DocTestRunner*'s output checker, and the results are formatted by the `DocTestRunner.report_*()` methods.

summarize(*verbose=None*)
Print a summary of all the test cases that have been run by this DocTestRunner, and return a *named tuple* `TestResults(failed, attempted)`.

The optional *verbose* argument controls how detailed the summary is. If the verbosity is not specified, then the *DocTestRunner*'s verbosity is used.

OutputChecker objects

class doctest.**OutputChecker**
A class used to check the whether the actual output from a doctest example matches the expected output. *OutputChecker* defines two methods: *check_output()*, which compares a given pair of outputs, and returns `True` if they match; and *output_difference()*, which returns a string describing the differences between two outputs.

OutputChecker defines the following methods:

check_output(*want*, *got*, *optionflags*)
Return `True` iff the actual output from an example (*got*) matches the expected output (*want*). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See section *Flags opcionais* for more information about option flags.

output_difference(*example*, *got*, *optionflags*)
Return a string describing the differences between the expected output for a given example (*example*) and the actual output (*got*). *optionflags* is the set of option flags used to compare *want* and *got*.

27.3.7 Depuração

Doctest provides several mechanisms for debugging doctest examples:

- Several functions convert doctests to executable Python programs, which can be run under the Python debugger, *pdb*.
- The *DebugRunner* class is a subclass of *DocTestRunner* that raises an exception for the first failing example, containing information about that example. This information can be used to perform post-mortem debugging on the example.
- The *unittest* cases generated by *DocTestSuite()* support the *debug()* method defined by *unittest.TestCase*.
- You can add a call to *pdb.set_trace()* in a doctest example, and you'll drop into the Python debugger when that line is executed. Then you can inspect current values of variables, and so on. For example, suppose `a.py` contains just this module docstring:

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print(x+3)
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

Then an interactive Python session may look like this:

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
  1     def g(x):
  2         print(x+3)
  3  ->     import pdb; pdb.set_trace()
[EOF]
(Pdb) p x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
  1     def f(x):
  2  ->     g(x*2)
[EOF]
(Pdb) p x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

Functions that convert doctests to Python code, and possibly run the synthesized code under the debugger:

`doctest.`**`script_from_examples`**`(`*s*`)`

Convert text with examples to a script.

Argument *s* is a string containing doctest examples. The string is converted to a Python script, where doctest examples in *s* are converted to regular code, and everything else is converted to Python comments. The generated script is returned as a string. For example,

```
import doctest
print(doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print(x+y)
```

(continua na próxima página)

(continuação da página anterior)

```
    3
"""))
```

displays:

```
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print(x+y)
# Expected:
## 3
```

This function is used internally by other functions (see below), but can also be useful when you want to transform an interactive Python session into a Python script.

doctest.**testsource**(*module*, *name*)

Convert the doctest for an object to a script.

Argument *module* is a module object, or dotted name of a module, containing the object whose doctests are of interest. Argument *name* is the name (within the module) of the object with the doctests of interest. The result is a string, containing the object's docstring converted to a Python script, as described for *script_from_examples()* above. For example, if module `a.py` contains a top-level function `f()`, then

```
import a, doctest
print(doctest.testsource(a, "a.f"))
```

prints a script version of function `f()`'s docstring, with doctests converted to code, and the rest placed in comments.

doctest.**debug**(*module*, *name*, *pm=False*)

Debug the doctests for an object.

The *module* and *name* arguments are the same as for function *testsource()* above. The synthesized Python script for the named object's docstring is written to a temporary file, and then that file is run under the control of the Python debugger, *pdb*.

A shallow copy of `module.__dict__` is used for both local and global execution context.

Optional argument *pm* controls whether post-mortem debugging is used. If *pm* has a true value, the script file is run directly, and the debugger gets involved only if the script terminates via raising an unhandled exception. If it does, then post-mortem debugging is invoked, via *pdb.post_mortem()*, passing the traceback object from the unhandled exception. If *pm* is not specified, or is false, the script is run under the debugger from the start, via passing an appropriate *exec()* call to *pdb.run()*.

doctest.**debug_src**(*src*, *pm=False*, *globs=None*)

Debug the doctests in a string.

This is like function *debug()* above, except that a string containing doctest examples is specified directly, via the *src* argument.

Optional argument *pm* has the same meaning as in function *debug()* above.

Optional argument *globs* gives a dictionary to use as both local and global execution context. If not specified, or `None`, an empty dictionary is used. If specified, a shallow copy of the dictionary is used.

The *DebugRunner* class, and the special exceptions it may raise, are of most interest to testing framework authors, and will only be sketched here. See the source code, and especially *DebugRunner*'s docstring (which is a doctest!) for more details:

class doctest.**DebugRunner**(*checker=None*, *verbose=None*, *optionflags=0*)
A subclass of *DocTestRunner* that raises an exception as soon as a failure is encountered. If an unexpected exception occurs, an *UnexpectedException* exception is raised, containing the test, the example, and the original exception. If the output doesn't match, then a *DocTestFailure* exception is raised, containing the test, the example, and the actual output.

For information about the constructor parameters and methods, see the documentation for *DocTestRunner* in section *Advanced API*.

There are two exceptions that may be raised by *DebugRunner* instances:

exception doctest.**DocTestFailure**(*test*, *example*, *got*)
An exception raised by *DocTestRunner* to signal that a doctest example's actual output did not match its expected output. The constructor arguments are used to initialize the attributes of the same names.

DocTestFailure defines the following attributes:

DocTestFailure.**test**
The *DocTest* object that was being run when the example failed.

DocTestFailure.**example**
The *Example* that failed.

DocTestFailure.**got**
The example's actual output.

exception doctest.**UnexpectedException**(*test*, *example*, *exc_info*)
An exception raised by *DocTestRunner* to signal that a doctest example raised an unexpected exception. The constructor arguments are used to initialize the attributes of the same names.

UnexpectedException defines the following attributes:

UnexpectedException.**test**
The *DocTest* object that was being run when the example failed.

UnexpectedException.**example**
The *Example* that failed.

UnexpectedException.**exc_info**
A tuple containing information about the unexpected exception, as returned by *sys.exc_info()*.

27.3.8 Soapbox

As mentioned in the introduction, *doctest* has grown to have three primary uses:

1. Checking examples in docstrings.
2. Regression testing.
3. Executable documentation / literate testing.

These uses have different requirements, and it is important to distinguish them. In particular, filling your docstrings with obscure test cases makes for bad documentation.

When writing a docstring, choose docstring examples with care. There's an art to this that needs to be learned—it may not be natural at first. Examples should add genuine value to the documentation. A good example can often be worth many words. If done with care, the examples will be invaluable for your users, and will pay back the time it takes to collect them many times over as the years go by and things change. I'm still amazed at how often one of my *doctest* examples stops working after a "harmless" change.

Doctest also makes an excellent tool for regression testing, especially if you don't skimp on explanatory text. By interleaving prose and examples, it becomes much easier to keep track of what's actually being tested, and why. When a test

fails, good prose can make it much easier to figure out what the problem is, and how it should be fixed. It's true that you could write extensive comments in code-based testing, but few programmers do. Many have found that using doctest approaches instead leads to much clearer tests. Perhaps this is simply because doctest makes writing prose a little easier than writing code, while writing comments in code is a little harder. I think it goes deeper than just that: the natural attitude when writing a doctest-based test is that you want to explain the fine points of your software, and illustrate them with examples. This in turn naturally leads to test files that start with the simplest features, and logically progress to complications and edge cases. A coherent narrative is the result, instead of a collection of isolated functions that test isolated bits of functionality seemingly at random. It's a different attitude, and produces different results, blurring the distinction between testing and explaining.

Regression testing is best confined to dedicated objects or files. There are several options for organizing tests:

- Write text files containing test cases as interactive examples, and test the files using *testfile()* or *DocFileSuite()*. This is recommended, although is easiest to do for new projects, designed from the start to use doctest.
- Define functions named `_regrtest_topic` that consist of single docstrings, containing test cases for the named topics. These functions can be included in the same file as the module, or separated out into a separate test file.
- Define a `__test__` dictionary mapping from regression test topics to docstrings containing test cases.

When you have placed your tests in a module, the module can itself be the test runner. When a test fails, you can arrange for your test runner to re-run only the failing doctest while you debug the problem. Here is a minimal example of such a test runner:

```
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.FAIL_FAST
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            obj = globals()[name]
        else:
            obj = __test__[name]
        doctest.run_docstring_examples(obj, globals(), name=name,
                                       optionflags=flags)
    else:
        fail, total = doctest.testmod(optionflags=flags)
        print("{} failures out of {} tests".format(fail, total))
```

27.4 `unittest` — Framework de Testes Unitários

Source code: Lib/unittest/__init__.py

(Caso já estejas familiarizado com os conceitos básicos de testes, poderás querer ignorar a lista de métodos assertivos 1.)

A estrutura de teste de unitários *unittest* foi originalmente inspirada no JUnit e tem um sabor semelhante contendo as principais estruturas de teste de unidades existentes em outras linguagens. Ele suporta a automação de teste, compartilhamento de configuração e código de desligamento para testes, agregação de testes em coleções e independência dos testes da estrutura de relatórios.

Para conseguir isso, o módulo *unittest* suporta alguns conceitos importantes de forma orientada a objetos:

equipamento de teste Um *test fixture* representa a preparação necessária para executar um ou mais testes e quaisquer ações de limpeza associadas. Isso pode envolver, por exemplo, a criação de bancos de dados temporários ou de proxy, a criação de diretórios ou a inicialização de processo no servidor.

caso de teste Um *test case* é uma unidade de teste individual. O mesmo verifica uma resposta específica a um determinado conjunto de entradas. O *unittest* fornece uma classe base, *TestCase*, que pode ser usada para criar novos casos de teste.

Suíte de Testes Uma *test suite* é uma coleção de casos de teste, conjuntos de teste ou ambos. O mesmo é usado para agregar testes que devem ser executados juntos.

test runner Um *test runner* é um componente que orquestra a execução de testes e fornece o resultado para o usuário. O runner pode usar uma interface gráfica, uma interface textual ou retornar um valor especial para indicar os resultados da execução dos testes.

Ver também:

Módulo *doctest* Outro módulo de suporte a testes com um sabor muito diferente.

Simple Smalltalk Testing: With Patterns O documento original de Kent Beck sobre estruturas de teste usando o padrão compartilhado por *unittest*.

pytest É um framework externo do unittest com uma sintaxe mais leve para escrever testes. Por exemplo, `assert func(10) == 42`.

The Python Testing Tools Taxonomy Uma extensa lista de ferramentas para testar código Python, incluindo estruturas de teste funcionais e bibliotecas de objetos simulados.

Testing in Python Mailing List Um grupo de interesse especial para discussão de testes e ferramentas de teste, em Python.

O script `Tools/unittestgui/unittestgui.py` na distribuição fonte do Python é uma ferramenta GUI para descobrimento e execução de teste. A intenção é facilitar o uso para aqueles que são novos no teste unitário. Para ambientes de produção é recomendado que os testes sejam executados por um sistema e integração contídua, como o Buildbot, Jenkins ou Hudson.

27.4.1 Exemplo Básico

O módulo *unittest* fornece um conjunto amplo de ferramentas para a construção e execução de testes. Esta seção demonstra que um pequeno subconjunto das ferramentas é suficiente para atender às necessidades da maioria dos usuários.

Aqui temos um simples Script para testar três métodos de String:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

Para criar um testcase basta criar uma classe que estende de *unittest.TestCase*. Os três testes individuais são definidos com métodos cujos nomes começam com as letras `test`. Esta convenção na nomenclatura informa o runner a respeitos de quais métodos sãso, na verdade, testes.

O cerne de cada teste é a invocação de um método *assertEqual()* para verificar se há um resultado esperado; *assertTrue()* ou *assertFalse()* para verificar uma condição; ou *assertRaises()* para verificar se uma exceção específica será levantada. Esses métodos são usados ao invés de utilizar a expressão `assert` para que o runner de teste possa acumular todos os resultados do teste e produzir um relatório.

Os métodos *setUp()* e *tearDown()* permitem que você defina instruções que serão executadas antes e depois de cada método de teste. Eles são abordados em mais detalhes na seção organize-tests.

O bloco final mostra uma maneira simples de executar os testes. A função *unittest.main()* fornece uma interface de linha de comando para o Script de teste. Quando executado a partir da linha de comando, o Script acima produz uma saída que se parece com isso:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Passando a opção `-v` para o nosso Script de teste instruirá a função :func: *unittest.main* a habilitar um nível mais alto de verbosidade e produzirá a seguinte saída:

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Os exemplos acima mostram os recursos mais utilizados *unittest* que são suficientes para atender a muitas necessidades de testes diários. O restante da documentação explora o conjunto completo de recursos desde os primeiros princípios.

27.4.2 Interface de Linha de Comando

O módulo unittest pode ser usado diretamente da linha de comando para rodar testes de módulos, classes ou mesmo testes de métodos individuais:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

Você pode passar uma lista com qualquer combinação de nomes de módulos e nomes de classes ou métodos totalmente qualificados.

Os módulos de teste podem ser especificados por caminhos de arquivo também:

```
python -m unittest tests/test_something.py
```

Isso permite que você use o auto-completar do console para especificar o módulo de teste. O arquivo especificado precisa ser "importável" como um módulo. O caminho é convertido para um nome de módulo ao remover a extensão *.py* e conversando os separadores do caminho em .. Se você quer executar um arquivo de teste que não é importável como um módulo, você deve executar o arquivo diretamente.

Você pode executar os testes com mais detalhes (maior verbosidade) ao usar a flag -v

```
python -m unittest -v test_module
```

Quando executado sem argumentos *Test Discovery* é iniciado:

```
python -m unittest
```

Para uma lista de todas as opções de linha de comando:

```
python -m unittest -h
```

Alterado na versão 3.2: Em versões mais antigas, era possível de executar apenas testes de métodos inviduais e não de módulos ou classes.

Opções de linha de comando

unittest suporta as seguintes opções de linha de comando:

-b, --buffer
Os streams da saída padrão e do erro padrão são carregados durante a execção do teste. A saída de um teste que passou é descartada. A saída geralmente é mostrada quando um teste falha e é adicionada às mensagens de falha.

-c, --catch
`Control-C` durante a execução do teste aguarda até que o teste corrente termine e a partir disso mostra todos os resultados até o momento. Um segundo `Control-C` invoca uma exceção *`KeyboardInterrupt`* comum.

Veja *Signal Handling* para as funções que provêm essa funcionalidade.

-f, --failfast
Parar a execução do teste no primeiro erro ou falha.

-k
Somente execute métodos de teste e classes que combinem com o padrão ou substring. Essa opção pode ser utilizada várias vezes, em cada caso todos os testes que combinam com o padrão dado serão incluídos.

Padrões que contém um caractere curinga (`*`) são combinados com os testes pelo método *`fnmatch.fnmatchcase()`*; caso contrário, é utilizada uma combinação simples de substrings, diferenciando-se letras maiúsculas e minúsculas.

Padrões são combinados com o nome completo qualificado do método de teste no formato que ele é importado pelo carregador.

Por exemplo, `-k foo` combina com `foo_tests.SomeTest.test_something`, `bar_tests.SomeTest.test_foo`, mas não com `bar_tests.FooTest.test_something`.

--locals
Mostra variáveis locais no traceback.

Novo na versão 3.2: As opções de linha de comando `-b`, `-c` e `-f` foram adicionadas.

Novo na versão 3.5: A opção de linha de comando `-locals`.

Novo na versão 3.7: A opção de linha de comando `-k`.

A linha de comando também pode ser usada para descobrir testes, para executar todos os testes de um projeto ou apenas de um subconjunto.

27.4.3 Test Discovery

Novo na versão 3.2.

O Unittest tem suporte para descobrimento simples de testes. Para serem compatíveis com o descobrimento de testes, todos os arquivos de teste devem ser módulos ou pacotes (incluindo *pacotes de espaço de nomes<namespace package>*) importáveis a partir do diretório raiz do projeto (isso significa que os nomes dos arquivos devem ser identificadores válidos).

O descobrimento de testes é implementado no *TestLoader.discover()*, mas também pode ser utilizado a partir da linha de comando. O comando básico para uso é:

```
cd project_directory
python -m unittest discover
```

Nota: Como um atalho, `python -m uniitest` é o equivalente a `python -m uniitest discover`. Se você deseja passar argumentos para a descoberta de testes, o subcomando `discover` deve ser usado explicitamente.

O sub-comando `discover` (descubra) tem as seguintes opções:

-v, --verbose
Saída verbosa

-s, --start-directory directory
Diretório no qual se inicia o descobrimento (`.` por padrão)

-p, --pattern pattern
Padrão de texto para se descobrir os arquivos de teste (`test*.py` por padrão)

-t, --top-level-directory directory
Diretório raiz do projeto (diretório de início por padrão)

As opções *-s*, *-p* e *-t* podem ser passadas como argumentos posicionais nessa ordem. As duas linhas de comando seguintes são equivalentes:

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```

Além de aceitar caminhos, também é possível passar o nome de um pacote, como `myproject.subpackage.test`, como diretório de início. O nome do pacote que for passado será importado e sua localização no sistema de arquivos será utilizada como diretório de início.

Cuidado: O descobridor de testes importa os testes para carregá-los. Uma vez que o descobridor tiver encontrado todos os arquivos de teste a partir do diretório de início especificado, ele transforma os caminhos em nomes de pacotes para conseguir importá-los. Por exemplo, `foo/bar/baz.py` será importado como `foo.bar.baz`.

Se você possuir um pacote instalado globalmente e tentar executar o descobrimento em uma versão diferente deste mesmo pacote, a importação *pode* acontecer do lugar errado. Se isso acontecer, o descobridor de testes irá emitir um alerta e encerrar a execução.

Se você configurar o diretório de início como sendo um nome de pacote, não um caminho para um diretório, o descobridor irá assumir que qualquer local do qual ele importar é o local correto. Neste caso, nenhum alerta será emitido.

Módulos de testes e pacotes podem conter customizações no carregamento de testes utilizando *load_tests protocol*.

Alterado na versão 3.4: O descobrimento de testes possui suporte para *pacotes de espaço de nomes*.

27.4.4 Organizando código teste

O bloco básico de construção dos testes unitários são os *casos de teste* — cenários únicos que devem ser configurados e avaliados em sua corretude. No *unittest*, casos de teste são representados por instâncias *unittest.TestCase*. Para criar seus próprios casos de teste, você deve escrever subclasses de *TestCase* ou utilizar *FunctionTestCase*.

O código de teste em uma instância da classe *TestCase* deve ser completamente auto-contido, de maneira que ele possa ser executado isoladamente ou combinado, de forma arbitrária, com quaisquer outros casos de teste.

A mais simples subclasse de *TestCase* irá implementar um método de teste (i.e. um método cujo nome começa com `test`) para executar um teste:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

Perceba que, para testar algo, utilizamos um dos métodos `assert*()` da classe base *TestCase*. Se o teste falhar, uma exceção será levantada com uma mensagem de explicação e o módulo *unittest* irá considerar o resultado do caso de teste como uma *falha*. Quaisquer outras exceções serão tratadas como *erros*.

Os testes podem ser muitos, e as configurações podem ser repetitivas. Por sorte, temos como reaproveitar estas configurações implementando um método chamado *setUp()*, o qual será automaticamente chamado pelo framework de teste para cada teste único que executarmos:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

Nota: A ordem de execução dos testes será feita com base na ordenação dos nomes dos métodos de teste respeitando os critérios da ordenação de strings embutida.

Se o método *setUp()* levantar uma exceção durante a sua execução, o framework irá considerar que o teste sofreu um erro e o método de teste não será executado.

De maneira similar, pode-se definir um método *tearDown()* que limpa o ambiente após a execução do método de teste:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')
```

(continua na próxima página)

(continuação da página anterior)

```
    def tearDown(self):
        self.widget.dispose()
```

Se o método *setUp()* for bem sucedido, o método *tearDown()* será executado, independente do resultado do método de teste.

Tal ambiente de trabalho do código de teste é chamado de *definição de contexto de teste*. Uma nova instância TestCase é criada como uma única definição de contexto de teste usada para executar cada método de teste. Portanto, os métodos *setUp()*, *tearDown()*, e `__init__()` serão chamados uma vez por teste.

É recomendado utilizar implementações de TestCase para agrupar testes de acordo com as funcionalidades que eles testam. *unittest* disponibiliza um mecanismo para isso: a *suíte de testes*, representada pela classe *TestSuite* do módulo *unittest*. Em grande parte dos casos, chamar *unittest.main()* é suficiente para coletar todos os casos de teste do módulo e executá-los para você.

Entretanto, caso você queira customizar a construção da sua suíte de testes, é possível fazê-la desta forma:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_widget_size'))
    suite.addTest(WidgetTestCase('test_widget_resize'))
    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

Você pode colocar as definições dos casos de teste e suíte de testes em um mesmo módulo que contém o código a ser testado (tal como `widget.py`), mas existem várias vantagens ao colocar o código dos testes em um módulo separado, como `test_widget.py`:

- O módulo de teste pode ser executado de maneira isolada a partir da linha de comando.
- O código de teste pode ser mais facilmente separado do código a ser entregue.
- Há uma menor tentação para modificar o código de teste para que ele se adeque ao código testado sem que haja uma boa razão.
- O código de teste deve ser modificado muito menos frequentemente do que o código que ele testa.
- Código testado pode ser reformuladomais facilmente
- Testes para módulos escritos em C devem ser, obrigatoriamente, colocados em módulos separados, então por que não manter a consistência?
- Se as estratégias de teste mudarem, não há a necessidade de mudar o código-fonte.

27.4.5 Reutilizando códigos de teste antigos

Alguns usuários irão encontrar antigos códigos de teste que eles gostariam de executar com *unittest* sem converter cada função antiga para uma subclasse de *TestCase*.

Por isso, *unittest* contém uma classe *FunctionTestCase*. Esta subclasse de *TestCase* pode ser utilizada para englobar funções de teste existentes. Funções de definição de estado inicial e final (set-up e tear-down) também podem ser fornecidas.

Dadas as seguintes funções de teste:

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

é possível criar uma instância de caso de teste, com métodos opcionais de configuração inicial e final (set-up e tear-down), como mostrado a seguir:

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

Nota: Apesar da classe *FunctionTestCase* poder ser utilizada para converter rapidamente um teste existente em um teste do módulo *unittest*, esta abordagem não é recomendada. Investir tempo para configurar corretamente uma subclasse de *TestCase* irá tornar futuras refatorações infinitamente mais fáceis.

Em alguns casos, os testes existentes podem ter sido escritos utilizando o módulo *doctest*. Se for o caso, *doctest* contém a classe `DocTestSuite` que pode construir, automaticamente, instâncias *unittest.TestSuite* a partir dos testes baseados em *doctest*.

27.4.6 Ignorando testes e falhas esperadas

Novo na versão 3.1.

Unittest suporta ignorar métodos de teste individuais e, até mesmo, classes de teste inteiras. Além disso, há suporte para a marcação de um teste como uma "falha esperada", um teste que está incorreto e irá falhar, mas não deve ser considerado como uma falha no *TestResult*.

Ignorar um teste é simplesmente uma questão de utilizar o *decorador* *skip()* ou uma de suas variantes condicionais, chamando *TestCase.skipTest()* em um *setUp()* ou método de teste, ou levantando *SkipTest* diretamente.

Ignorar se parece basicamente com:

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass

    def test_maybe_skipped(self):
        if not external_resource_available():
            self.skipTest("external resource not available")
        # test code that depends on the external resource
        pass
```

Esta é a saída da execução do exemplo acima em modo verboso:

```
test_format (__main__.MyTestCase) ... skipped 'not supported in this library version'
test_nothing (__main__.MyTestCase) ... skipped 'demonstrating skipping'
test_maybe_skipped (__main__.MyTestCase) ... skipped 'external resource not available'
test_windows_support (__main__.MyTestCase) ... skipped 'requires Windows'

----------------------------------------------------------------------
Ran 4 tests in 0.005s

OK (skipped=4)
```

Classes podem ser puladas assim como métodos:

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

TestCase.setUp() também pode ignorar o teste. Isso é útil quando um recurso que precisa ser configurado não está disponível.

Falhas esperadas usam o decorador *expectedFailure()*:

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

É simples ignorar com decoradores customizados, basta fazer um decorador que chama *skip()* no teste quando ele deve ser ignorado. Este decorador ignora o teste a não ser que o objeto fornecido tenha um determinado atributo:

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

Os decoradores e exceção seguintes ignoram testes e falhas esperadas:

`@unittest.`**`skip`**`(`*reason*`)`
: Ignora incondicionalmente o teste decorado. *reason* deve descrever a razão pela qual o teste está sendo ignorado.

`@unittest.`**`skipIf`**`(`*condition*, *reason*`)`
: Ignora o teste decorado se *condition* for verdadeiro.

`@unittest.`**`skipUnless`**`(`*condition*, *reason*`)`
: Ignora o teste decorado a não ser que *condition* seja verdadeiro.

`@unittest.`**`expectedFailure`**
: Marca o teste como uma falha esperada. Se o teste falhar, ele será considerado um sucesso. Se o teste passar, ele será considerado uma falha.

`exception` `unittest.`**`SkipTest`**`(`*reason*`)`
: Esta exceção é levantada para ignorar um teste.

  Normalmente, você pode utilizar *TestCase.skipTest()* ou um dos decoradores para ignorar sem ter de levantar esta exceção diretamente.

Testes ignorados não terão seus métodos *setUp()* ou *tearDown()* executados. Classes ignoradas não terão seus métodos *setUpClass()* ou *tearDownClass()* executados. Módulos ignorados não terão seus métodos `setUpModule()` ou `tearDownModule()` executados.

27.4.7 Distinguindo iterações de teste utilizando subtestes

Novo na versão 3.4.

Quando existem pequenas diferenças entre os seus testes, por exemplo alguns parâmetros, unittest permite que você distinga-os dentro do corpo de um método de teste utilizando o gerenciador de contexto *subTest()*.

Por exemplo, o teste seguinte:

```
class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        Test that numbers between 0 and 5 are all even.
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

irá produzir a seguinte saída:

```
======================================================================
FAIL: test_even (__main__.NumbersTest) (i=1)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest) (i=3)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest) (i=5)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

Sem usar um subteste, a execução para depois da primeira falha e o erro será menos fácil de ser diagnosticado porque o valor de `i` não será mostrado:

```
======================================================================
FAIL: test_even (__main__.NumbersTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

27.4.8 Classes e funções

Esta seção descreve de maneira aprofundada a API do módulo *unittest*.

Casos de teste

class unittest.**TestCase**(*methodName='runTest'*)

Instâncias da classe *TestCase* representam unidades lógicas de teste no universo do *unittest*. Esta classe deve ser utilizada como classe base, com testes específicos sendo implementados por subclasses concretas. Esta classe implementa a interface requerida pelo executor de testes, para permitir o controle dos testes, e métodos que o código de teste pode utilizar para checar e reportar diversos tipos de falhas.

Cada instância da classe *TestCase* irá executar um único método base: o método chamado *methodName*. Em muitos casos de uso da classe *TestCase*, você não precisará modificar o *methodName* ou reimplementar o método `runTest()` padrão.

Alterado na versão 3.2: *TestCase* pode ser instanciada com sucesso sem fornecer um *methodName*. Isso torna mais fácil experimentar com *TestCase* a partir de um interpretador interativo.

Instâncias *TestCase* possuem três grupos de métodos: um grupo utilizado para executar o teste, outro utilizado pela implementação do teste para checar as condições e reportar falhas, e alguns métodos de investigação para permitir coletar dados sobre o teste em si.

Os métodos do primeiro grupo (que executam os testes) são:

setUp()

Método chamado para preparar a definição de contexto de teste. Chamado imediatamente após chamar o método de teste; exceto pelo *AssertionError* ou *SkipTest*, qualquer exceção levantada por este método será considerada um erro além de uma simples falha de teste. A implementação padrão não faz nada.

tearDown()

Método chamado imediatamente após o método de teste ter sido chamado e o resultado registrado. Este método é chamado mesmo se o método de teste tiver levantado uma exceção, então a implementação em subclasses deve ser feita com cautela ao verificar o estado interno. Qualquer exceção além de *AssertionError* ou *SkipTest* levantada por este método será considerado um erro adicional, e não uma simples falha de teste (que incrementaria o número total de erros no relatório final de testes). Este método será executado apenas se o método *setUp()* for bem sucedido, independente do resultado do método de teste. A implementação padrão não faz nada.

setUpClass()

Um método de classe chamado antes da execução dos testes de uma classe específica. O método `setUpClass` é chamado com a classe sendo o único argumento e deve ser decorada como um *classmethod()*:

```
@classmethod
def setUpClass(cls):
    ...
```

Veja *Classes e Módulos de Definição de Contexto* para mais detalhes.

Novo na versão 3.2.

tearDownClass()

Um método de classe chamado depois da execução dos testes de uma classe específica. O método `tearDownClass` é chamado com a classe sendo o único argumento e deve ser decorada como um *classmethod()*:

```
@classmethod
def tearDownClass(cls):
    ...
```

Veja *Classes e Módulos de Definição de Contexto* para mais detalhes.

Novo na versão 3.2.

run(*result=None*)
Executa o teste, registrando as informações no objeto resultado `TestResult`, passado como *result*. Se *result* for omitido, ou definido como `None`, um objeto resultado temporário é criado (chamando o método `defaultTestResult()`) e utilizado. O objeto resultado é retornado para quem chamou o método `run()`.

O mesmo efeito pode ser obtido ao chamar uma instância da classe `TestCase`.

Alterado na versão 3.3: Versões anteriores de `run` não retornavam o resultado. Nem chamando por uma instância.

skipTest(*reason*)
Ao se executar durante um método de teste ou `setUp()`, pula o teste em execução. Veja *Ignorando testes e falhas esperadas* para mais informações.

Novo na versão 3.1.

subTest(*msg=None*, ***params*)
Retorna um gerenciador de contexto que executa, como subteste, o bloco de código englobado. *msg* e *params* são valores opcionais e arbitrários que são mostrados sempre quando um subteste falha, permitindo identificá-los claramente.

Um caso de teste pode conter inúmeras declarações de subteste e elas podem ser aninhadas de forma arbitrária.

Veja *Distinguindo iterações de teste utilizando subtestes* para mais informações.

Novo na versão 3.4.

debug()
Executa o teste sem coletar o resultado. Permite propagar exceções levantadas pelo teste e pode ser utilizado para oferecer suporte aos testes sob um depurador.

A classe `TestCase` oferece diversos métodos de asserção para checar e reportar falhas. A tabela a seguir lista os métodos mais utilizados (veja as tabelas abaixo para mais métodos de asserção).

Método	Avalia se	Novo em
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

Todos os métodos de asserção aceitam um argumento *msg* que, se especificado, é utilizado como a mensagem de erro em caso de falha (veja também `longMessage`). Note que o argumento nomeado *msg* pode ser pas-

sado para *assertRaises()*, *assertRaisesRegex()*, *assertWarns()* e *assertWarnsRegex()* apenas quando eles são utilizados como gerenciador de contexto.

assertEqual(*first*, *second*, *msg=None*)
Testa se *first* e *second* são iguais. Se o resultado da comparação não indicar igualdade, o teste irá falhar.

Além disso, se *first* e *second* são exatamente do mesmo tipo e são uma lista, tupla, dict, set, frozenset, str, ou qualquer outro tipo que é registrado na subclasse com *addTypeEqualityFunc()*, a função de igualdade específica do tipo é será chamada para gerar uma mensagem de erro mais útil (veja também *lista de métodos específicos por tipo*)

Alterado na versão 3.1: Adição da chamada automática da função específica por tipo.

Alterado na versão 3.2: método *assertMultiLineEqual()* adicionado como função de igualdade padrão para o tipo string.

assertNotEqual(*first*, *second*, *msg=None*)
Testa se *first* e *second* não são iguais. Se o resultado da comparação indicar igualdade, o teste irá falhar.

assertTrue(*expr*, *msg=None*)
assertFalse(*expr*, *msg=None*)
Testa se a expressão *expr* é verdadeira (ou falsa).

Note que isso é equivalente a `bool(expr) is True` e não a `expr is True` (use `assertIs(expr, True)` neste último caso). Este método também deve ser evitado quando outros métodos mais específicos estão disponíveis (e.g. `assertEqual(a, b)` no lugar de `assertTrue(a == b)`), já que estes podem ter uma melhor mensagem de erro em caso de falha.

assertIs(*first*, *second*, *msg=None*)
assertIsNot(*first*, *second*, *msg=None*)
Testa se *first* e *second* são avaliados (ou não são avaliados) para o mesmo objeto.

Novo na versão 3.1.

assertIsNone(*expr*, *msg=None*)
assertIsNotNone(*expr*, *msg=None*)
Testa se *expr* é (ou não é) `None`.

Novo na versão 3.1.

assertIn(*member*, *container*, *msg=None*)
assertNotIn(*member*, *container*, *msg=None*)
Testa se *member* está (ou não está) em *container*.

Novo na versão 3.1.

assertIsInstance(*obj*, *cls*, *msg=None*)
assertNotIsInstance(*obj*, *cls*, *msg=None*)
Testa se *obj* é (ou não é) uma instância de *cls* (que pode ser uma classe ou uma tupla de classes, como suportado pela função *isinstance()*). Para checar pelo tipo exato, use *assertIs(type(obj), cls)*.

Novo na versão 3.2.

Também é possível checar a produção de exceções, avisos e mensagens de log usando os seguintes métodos:

Método	Avalia se	Novo em
*assertRaises(exc, fun, *args, **kwds)*	`fun(*args, **kwds)` levanta *exc*	
*assertRaisesRegex(exc, r, fun, *args, **kwds)*	`fun(*args, **kwds)` levanta *exc* e a mensagem casa com a expressão regular *r*	3.1
*assertWarns(warn, fun, *args, **kwds)*	`fun(*args, **kwds)` levanta *warn*	3.2
*assertWarnsRegex(warn, r, fun, *args, **kwds)*	`fun(*args, **kwds)` levanta *warn* e a mensagem casa com a expressão regular *r*	3.2
assertLogs(logger, level)	O bloco `with` registra logs no *logger* com pelo menos um nível *level*	Módulos para processamento de XML

assertRaises(*exception*, *callable*, **args*, ***kwds*)
assertRaises(*exception*, *, *msg=None*)

Testa se uma exceção é levantada quando *callable* é chamado com quaisquer argumentos nomeados ou posicionais que também são passados para *assertRaises()*. O teste passa se *exception* for levantada e falha se outra exceção ou nenhuma exceção for levantada.

Se somente os argumentos *exception* e, possivelmente, *msg* forem passados, retorna um gerenciador de contexto para que o código sob teste possa ser escrito de forma embutida ao invés de ser definido como uma função:

```
with self.assertRaises(SomeException):
    do_something()
```

Quando utilizado como gerenciador de contexto, *assertRaises()* aceita o argumento nomeado adicional *msg*.

O gerenciador de contexto irá armazenar o objeto exceção capturado no seu atributo `exception`. Isso pode ser útil se a intenção for realizar testes adicionais na exceção levantada:

```
with self.assertRaises(SomeException) as cm:
    do_something()

the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

Alterado na versão 3.1: Adicionada a possibilidade de se utilizar *assertRaises()* como gerenciador de contexto.

Alterado na versão 3.2: Adicionado o atributo `exception`.

Alterado na versão 3.3: Adicionado o argumento nomeado *msg* quando utilizado como gerenciador de contexto.

assertRaisesRegex(*exception*, *regex*, *callable*, **args*, ***kwds*)
assertRaisesRegex(*exception*, *regex*, *, *msg=None*)

Similar ao *assertRaises()* mas também testa se *regex* casa com a representação em string da exceção levantada. *regex* pode ser um objeto expressão regular ou uma string contendo uma expressão regular compatível com o uso pela função *re.search()*. Exemplos:

```
self.assertRaisesRegex(ValueError, "invalid literal for.*XYZ'$",
                       int, 'XYZ')
```

or:

```
with self.assertRaisesRegex(ValueError, 'literal'):
   int('XYZ')
```

Novo na versão 3.1: Adicionado com o nome `assertRaisesRegexp`.

Alterado na versão 3.2: Renomeado para *`assertRaisesRegex()`*.

Alterado na versão 3.3: Adicionado o argumento nomeado *msg* quando utilizado como gerenciador de contexto.

assertWarns(*warning*, *callable*, **args*, ***kwds*)
assertWarns(*warning*, ***, *msg=None*)

Testa se o aviso *warning* é acionado quando *callable* é chamado com quaisquer argumentos nomeados ou posicionais que também são passados para *`assertWarns()`*. O teste passa se *warning* for acionado e falha se não for. Qualquer exceção é considerada como falha. Para capturar qualquer aviso presente em um grupo de avisos, uma tupla contendo as classes de aviso podem ser passadas como *warning*.

Se apenas os argumentos *warning* e, possivelmente, *msg* forem passados, retorna um gerenciador de contexto para que o código sob teste possa ser escrito de forma embutida ao invés de ser definido como uma função:

```
with self.assertWarns(SomeWarning):
    do_something()
```

Quando usado como gerenciador de contexto, *`assertWarns()`* aceita o argumento nomeado adicional *msg*.

O gerenciador de contexto irá armazenar o objeto de aviso capturado no atributo `warning` e a linha de código fonte que acionou o aviso nos atributos `filename` e `lineno`. Isso pode ser útil se a intenção for executar verificações adicionais no aviso capturado:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()

self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)
```

Este método funciona independente dos filtros de aviso configurados no momento em que é chamado.

Novo na versão 3.2.

Alterado na versão 3.3: Adicionado o argumento nomeado *msg* quando utilizado como gerenciador de contexto.

assertWarnsRegex(*warning*, *regex*, *callable*, **args*, ***kwds*)
assertWarnsRegex(*warning*, *regex*, ***, *msg=None*)

Similar ao *`assertWarns()`* mas também testa se *regex* casa com a mensagem do aviso acionado. *regex* pode ser um objeto expressão regular ou uma string contendo uma expressão regular compatível para uso pela função *`re.search()`*. Exemplo:

```
self.assertWarnsRegex(DeprecationWarning,
                      r'legacy_function\(\) is deprecated',
                      legacy_function, 'XYZ')
```

or:

```
with self.assertWarnsRegex(RuntimeWarning, 'unsafe frobnicating'):
    frobnicate('/etc/passwd')
```

Novo na versão 3.2.

Alterado na versão 3.3: Adicionado o argumento nomeado *msg* quando utilizado como gerenciador de contexto.

assertLogs(*logger=None*, *level=None*)

Um gerenciador de contexto para testar se pelo menos uma mensagem é inserida em *logger* ou em um de seus filhos, com pelo menos um nível *level*.

Se passado, *logger* deve ser um objeto da classe *logging.Logger* ou um objeto da classe *str* com o nome do registrador de logs. O padrão é o registrador de log raíz, que irá capturar todas as mensagens.

Se passado, *level* deve ser um nível de log numérico ou a string equivalente (por exemplo, `"ERROR"` ou `logging.ERROR`). O padrão é `logging.INFO`.

O teste passa se pelo menos uma das mensagens emitidas dentro do bloco `with` casa com as condições dadas por *logger* e *level*, falhando caso contrário.

O objeto retornado pelo gerenciador de contexto é um registro auxiliar que mantém os rastros das mensagens de log capturadas de acordo com os critérios dados. Ele possui dois atributos:

records
Uma lista de objetos de log da classe *logging.LogRecord* que foram compatíveis com os critérios dados.

output
Uma lista de objetos da classe *str* com a saída formatada das mensagens de log compatíveis com os critérios dados.

Exemplo:

```
with self.assertLogs('foo', level='INFO') as cm:
   logging.getLogger('foo').info('first message')
   logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

Novo na versão 3.4.

Existem também outros métodos usados para executar verificações mais específicas, como:

Método	Avalia se	Novo em
assertAlmostEqual(a, b)	`round(a-b, 7) == 0`	
assertNotAlmostEqual(a, b)	`round(a-b, 7) != 0`	
assertGreater(a, b)	`a > b`	3.1
assertGreaterEqual(a, b)	`a >= b`	3.1
assertLess(a, b)	`a < b`	3.1
assertLessEqual(a, b)	`a <= b`	3.1
assertRegex(s, r)	`r.search(s)`	3.1
assertNotRegex(s, r)	`not r.search(s)`	3.2
assertCountEqual(a, b)	*a* e *b* possuem os mesmos elementos na mesma quantidade, independente da sua ordem.	3.2

assertAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)

assertNotAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)
Testa se *first* e *second* são (ou não são) aproximadamente iguais considerando a diferença entre eles, arredondando para o número de casas decimais dado (7 por padrão), e comparando a zero. Note que estes métodos arredondam os valores considerando o número de casas decimais (i.e. como a função `round()`) e não o número de algarismos significativos.

Se *delta* é fornecido no lugar de *places*, a diferença entre *first* e *second* deve ser menor ou igual a (ou maior que) *delta*.

Passar *delta* e *places* ao mesmo tempo levanta a exceção `TypeError`.

Alterado na versão 3.2: `assertAlmostEqual()` considera, automaticamente, objetos quase iguais que possuem a comparação de igualdade dada como verdadeira. `assertNotAlmostEqual()` falha automaticamente se os objetos possuem a comparação de igualdade dada como verdadeira. Adicionado o argumento nomeado *delta*.

assertGreater(*first*, *second*, *msg=None*)
assertGreaterEqual(*first*, *second*, *msg=None*)
assertLess(*first*, *second*, *msg=None*)
assertLessEqual(*first*, *second*, *msg=None*)
Testa que *first* é respectivamente>, >=, < ou <= que *second*, dependendo do nome do método. Se não for, o teste irá falhar:

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

Novo na versão 3.1.

assertRegex(*text*, *regex*, *msg=None*)
assertNotRegex(*text*, *regex*, *msg=None*)
Testa que uma procura por *regex* dá match (ou não dá match) com o *text*. Em caso de falha, a mensagem de erro irá incluir o padrão e o *text* (ou o padrão e a parte do *text* que inesperadamente deu match). *regex* pode ser um objeto de expressão regular ou uma string contendo uma expressão regular adequada para uso por `re.search()`.

Novo na versão 3.1: Adicionada abaixo do nome `assertRegexpMatches`.

Alterado na versão 3.2: O método `assertRegexpMatches()` foi renomeado para `assertRegex()`.

Novo na versão 3.2: `assertNotRegex()`.

Novo na versão 3.5: O nome `assertNotRegexpMatches` é um apelido descontinuado para `assertNotRegex()`.

assertCountEqual(*first*, *second*, *msg=None*)
Test that sequence *first* contains the same elements as *second*, regardless of their order. When they don't, an error message listing the differences between the sequences will be generated.

Elementos duplicados *não* são ignorados ao comparar *first* e *second*. É verificado se cada elemento tem a mesma contagem em ambas sequências. Equivalente a: `assertEqual(Counter(list(first)), Counter(list(second)))`, mas também funciona com sequências de objetos não hasheáveis.

Novo na versão 3.2.

The `assertEqual()` method dispatches the equality check for objects of the same type to different type-specific methods. These methods are already implemented for most of the built-in types, but it's also possible to register new methods using `addTypeEqualityFunc()`:

addTypeEqualityFunc(*typeobj*, *function*)
Registers a type-specific method called by `assertEqual()` to check if two objects of exactly the same *typeobj* (not subclasses) compare equal. *function* must take two positional arguments and a third msg=None

keyword argument just as `assertEqual()` does. It must raise `self.failureException(msg)` when inequality between the first two parameters is detected – possibly providing useful information and explaining the inequalities in details in the error message.

Novo na versão 3.1.

The list of type-specific methods automatically used by `assertEqual()` are summarized in the following table. Note that it's usually not necessary to invoke these methods directly.

Método	Usado para comparar	Novo em
`assertMultiLineEqual(a, b)`	strings	3.1
`assertSequenceEqual(a, b)`	sequências	3.1
`assertListEqual(a, b)`	listas	3.1
`assertTupleEqual(a, b)`	tuplas	3.1
`assertSetEqual(a, b)`	sets ou frozensets	3.1
`assertDictEqual(a, b)`	dicionários	3.1

assertMultiLineEqual(*first*, *second*, *msg=None*)

Test that the multiline string *first* is equal to the string *second*. When not equal a diff of the two strings highlighting the differences will be included in the error message. This method is used by default when comparing strings with `assertEqual()`.

Novo na versão 3.1.

assertSequenceEqual(*first*, *second*, *msg=None*, *seq_type=None*)

Tests that two sequences are equal. If a *seq_type* is supplied, both *first* and *second* must be instances of *seq_type* or a failure will be raised. If the sequences are different an error message is constructed that shows the difference between the two.

This method is not called directly by `assertEqual()`, but it's used to implement `assertListEqual()` and `assertTupleEqual()`.

Novo na versão 3.1.

assertListEqual(*first*, *second*, *msg=None*)
assertTupleEqual(*first*, *second*, *msg=None*)

Tests that two lists or tuples are equal. If not, an error message is constructed that shows only the differences between the two. An error is also raised if either of the parameters are of the wrong type. These methods are used by default when comparing lists or tuples with `assertEqual()`.

Novo na versão 3.1.

assertSetEqual(*first*, *second*, *msg=None*)

Tests that two sets are equal. If not, an error message is constructed that lists the differences between the sets. This method is used by default when comparing sets or frozensets with `assertEqual()`.

Fails if either of *first* or *second* does not have a `set.difference()` method.

Novo na versão 3.1.

assertDictEqual(*first*, *second*, *msg=None*)

Test that two dictionaries are equal. If not, an error message is constructed that shows the differences in the dictionaries. This method will be used by default to compare dictionaries in calls to `assertEqual()`.

Novo na versão 3.1.

Finally the `TestCase` provides the following methods and attributes:

fail(*msg=None*)

Signals a test failure unconditionally, with *msg* or `None` for the error message.

failureException
This class attribute gives the exception raised by the test method. If a test framework needs to use a specialized exception, possibly to carry additional information, it must subclass this exception in order to "play fair" with the framework. The initial value of this attribute is *AssertionError*.

longMessage
This class attribute determines what happens when a custom failure message is passed as the msg argument to an assertXYY call that fails. `True` is the default value. In this case, the custom message is appended to the end of the standard failure message. When set to `False`, the custom message replaces the standard message.

The class setting can be overridden in individual test methods by assigning an instance attribute, self.longMessage, to `True` or `False` before calling the assert methods.

The class setting gets reset before each test call.

Novo na versão 3.1.

maxDiff
This attribute controls the maximum length of diffs output by assert methods that report diffs on failure. It defaults to 80*8 characters. Assert methods affected by this attribute are *assertSequenceEqual()* (including all the sequence comparison methods that delegate to it), *assertDictEqual()* and *assertMultiLineEqual()*.

Setting `maxDiff` to `None` means that there is no maximum length of diffs.

Novo na versão 3.2.

Frameworks de teste podem usar os seguintes métodos para coletar informações sobre o teste:

countTestCases()
Retorna o número de testes representados por esse objeto de teste. Para instâncias *TestCase* será sempre"1".

defaultTestResult()
Return an instance of the test result class that should be used for this test case class (if no other result instance is provided to the *run()* method).

For *TestCase* instances, this will always be an instance of *TestResult*; subclasses of *TestCase* should override this as necessary.

id()
Retorna uma string identificando o caso de teste específico. Geralmente é o nome completo do método do teste, incluindo o módulo e o nome da classe.

shortDescription()
Retorna uma descrição do teste ou"None" se nenhuma descrição for fornecida. A implementação padrão desse método retorna a primeira linha da docstring do método do teste, se disponível, ou `None`.

Alterado na versão 3.1: In 3.1 this was changed to add the test name to the short description even in the presence of a docstring. This caused compatibility issues with unittest extensions and adding the test name was moved to the *TextTestResult* in Python 3.2.

addCleanup(*function*, **args*, ***kwargs*)
Add a function to be called after *tearDown()* to cleanup resources used during the test. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into *addCleanup()* when they are added.

If *setUp()* fails, meaning that *tearDown()* is not called, then any cleanup functions added will still be called.

Novo na versão 3.1.

doCleanups()
This method is called unconditionally after *tearDown()*, or after *setUp()* if *setUp()* raises an exception.

It is responsible for calling all the cleanup functions added by *addCleanup()*. If you need cleanup functions to be called *prior* to *tearDown()* then you can call *doCleanups()* yourself.

doCleanups() pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

Novo na versão 3.1.

class unittest.**FunctionTestCase**(*testFunc*, *setUp=None*, *tearDown=None*, *description=None*)
This class implements the portion of the *TestCase* interface which allows the test runner to drive the test, but does not provide the methods which test code can use to check and report errors. This is used to create test cases using legacy test code, allowing it to be integrated into a *unittest*-based test framework.

Deprecated aliases

For historical reasons, some of the *TestCase* methods had one or more aliases that are now deprecated. The following table lists the correct names along with their deprecated aliases:

Método	Apelido descontinuado	Apelido descontinuado
assertEqual()	failUnlessEqual	assertEquals
assertNotEqual()	failIfEqual	assertNotEquals
assertTrue()	failUnless	assert_
assertFalse()	failIf	
assertRaises()	failUnlessRaises	
assertAlmostEqual()	failUnlessAlmostEqual	assertAlmostEquals
assertNotAlmostEqual()	failIfAlmostEqual	assertNotAlmostEquals
assertRegex()		assertRegexpMatches
assertNotRegex()		assertNotRegexpMatches
assertRaisesRegex()		assertRaisesRegexp

Obsoleto desde a versão 3.1: The fail* aliases listed in the second column have been deprecated.

Obsoleto desde a versão 3.2: The assert* aliases listed in the third column have been deprecated.

Obsoleto desde a versão 3.2: `assertRegexpMatches` and `assertRaisesRegexp` have been renamed to *assertRegex()* and *assertRaisesRegex()*.

Obsoleto desde a versão 3.5: The `assertNotRegexpMatches` name is deprecated in favor of *assertNotRegex()*.

Grouping tests

class unittest.**TestSuite**(*tests=()*)
This class represents an aggregation of individual test cases and test suites. The class presents the interface needed by the test runner to allow it to be run as any other test case. Running a *TestSuite* instance is the same as iterating over the suite, running each test individually.

If *tests* is given, it must be an iterable of individual test cases or other test suites that will be used to build the suite initially. Additional methods are provided to add test cases and suites to the collection later on.

TestSuite objects behave much like *TestCase* objects, except they do not actually implement a test. Instead, they are used to aggregate tests into groups of tests that should be run together. Some additional methods are available to add tests to *TestSuite* instances:

addTest(*test*)
Add a *TestCase* or *TestSuite* to the suite.

addTests(*tests*)
Add all the tests from an iterable of *TestCase* and *TestSuite* instances to this test suite.

This is equivalent to iterating over *tests*, calling *addTest()* for each element.

TestSuite shares the following methods with *TestCase*:

run(*result*)
Run the tests associated with this suite, collecting the result into the test result object passed as *result*. Note that unlike *TestCase.run()*, *TestSuite.run()* requires the result object to be passed in.

debug()
Run the tests associated with this suite without collecting the result. This allows exceptions raised by the test to be propagated to the caller and can be used to support running tests under a debugger.

countTestCases()
Return the number of tests represented by this test object, including all individual tests and sub-suites.

__iter__()
Tests grouped by a *TestSuite* are always accessed by iteration. Subclasses can lazily provide tests by overriding *__iter__()*. Note that this method may be called several times on a single suite (for example when counting tests or comparing for equality) so the tests returned by repeated iterations before *TestSuite.run()* must be the same for each call iteration. After *TestSuite.run()*, callers should not rely on the tests returned by this method unless the caller uses a subclass that overrides `TestSuite._removeTestAtIndex()` to preserve test references.

Alterado na versão 3.2: In earlier versions the *TestSuite* accessed tests directly rather than through iteration, so overriding *__iter__()* wasn't sufficient for providing tests.

Alterado na versão 3.4: In earlier versions the *TestSuite* held references to each *TestCase* after *TestSuite.run()*. Subclasses can restore that behavior by overriding `TestSuite._removeTestAtIndex()`.

In the typical usage of a *TestSuite* object, the *run()* method is invoked by a `TestRunner` rather than by the end-user test harness.

Carregando e executando testes

class unittest.**TestLoader**
The *TestLoader* class is used to create test suites from classes and modules. Normally, there is no need to create an instance of this class; the *unittest* module provides an instance that can be shared as *unittest.defaultTestLoader*. Using a subclass or instance, however, allows customization of some configurable properties.

Objetos da classe *TestLoader* possuem os seguintes atributos:

errors
A list of the non-fatal errors encountered while loading tests. Not reset by the loader at any point. Fatal errors are signalled by the relevant a method raising an exception to the caller. Non-fatal errors are also indicated by a synthetic test that will raise the original error when run.

Novo na versão 3.5.

Objetos da classe *TestLoader* possuem os seguintes métodos:

loadTestsFromTestCase(*testCaseClass*)
Return a suite of all test cases contained in the *TestCase*-derived `testCaseClass`.

A test case instance is created for each method named by *getTestCaseNames()*. By default these are the method names beginning with `test`. If *getTestCaseNames()* returns no methods, but the `runTest()` method is implemented, a single test case is created for that method instead.

loadTestsFromModule(*module*, *pattern=None*)
Return a suite of all test cases contained in the given module. This method searches *module* for classes derived from *TestCase* and creates an instance of the class for each test method defined for the class.

Nota: While using a hierarchy of *TestCase*-derived classes can be convenient in sharing fixtures and helper functions, defining test methods on base classes that are not intended to be instantiated directly does not play well with this method. Doing so, however, can be useful when the fixtures are different and defined in subclasses.

If a module provides a `load_tests` function it will be called to load the tests. This allows modules to customize test loading. This is the *load_tests protocol*. The *pattern* argument is passed as the third argument to `load_tests`.

Alterado na versão 3.2: Suporte para `load_tests` adicionado.

Alterado na versão 3.5: The undocumented and unofficial *use_load_tests* default argument is deprecated and ignored, although it is still accepted for backward compatibility. The method also now accepts a keyword-only argument *pattern* which is passed to `load_tests` as the third argument.

loadTestsFromName(*name*, *module=None*)
Return a suite of all test cases given a string specifier.

The specifier *name* is a "dotted name" that may resolve either to a module, a test case class, a test method within a test case class, a *TestSuite* instance, or a callable object which returns a *TestCase* or *TestSuite* instance. These checks are applied in the order listed here; that is, a method on a possible test case class will be picked up as "a test method within a test case class", rather than "a callable object".

For example, if you have a module `SampleTests` containing a *TestCase*-derived class `SampleTestCase` with three test methods (`test_one()`, `test_two()`, and `test_three()`), the specifier `'SampleTests.SampleTestCase'` would cause this method to return a suite which will run all three test methods. Using the specifier `'SampleTests.SampleTestCase.test_two'` would cause it to return a test suite which will run only the `test_two()` test method. The specifier can refer to modules and packages which have not been imported; they will be imported as a side-effect.

O método opcionalmente resolve o *nome* relativo ao *módulo* dado.

Alterado na versão 3.5: If an *ImportError* or *AttributeError* occurs while traversing *name* then a synthetic test that raises that error when run will be returned. These errors are included in the errors accumulated by self.errors.

loadTestsFromNames(*names*, *module=None*)
Similar to *loadTestsFromName()*, but takes a sequence of names rather than a single name. The return value is a test suite which supports all the tests defined for each name.

getTestCaseNames(*testCaseClass*)
Return a sorted sequence of method names found within *testCaseClass*; this should be a subclass of *TestCase*.

discover(*start_dir*, *pattern='test*.py'*, *top_level_dir=None*)
Find all the test modules by recursing into subdirectories from the specified start directory, and return a TestSuite object containing them. Only test files that match *pattern* will be loaded. (Using shell style pattern matching.) Only module names that are importable (i.e. are valid Python identifiers) will be loaded.

All test modules must be importable from the top level of the project. If the start directory is not the top level directory then the top level directory must be specified separately.

If importing a module fails, for example due to a syntax error, then this will be recorded as a single error and discovery will continue. If the import failure is due to *SkipTest* being raised, it will be recorded as a skip instead of an error.

If a package (a directory containing a file named `__init__.py`) is found, the package will be checked for a `load_tests` function. If this exists then it will be called `package.load_tests(loader, tests, pattern)`. Test discovery takes care to ensure that a package is only checked for tests once during an invocation, even if the load_tests function itself calls `loader.discover`.

If `load_tests` exists then discovery does *not* recurse into the package, `load_tests` is responsible for loading all tests in the package.

The pattern is deliberately not stored as a loader attribute so that packages can continue discovery themselves. *top_level_dir* is stored so `load_tests` does not need to pass this argument in to `loader.discover()`.

start_dir can be a dotted module name as well as a directory.

Novo na versão 3.2.

Alterado na versão 3.4: Modules that raise *SkipTest* on import are recorded as skips, not errors. Discovery works for *namespace packages*. Paths are sorted before being imported so that execution order is the same even if the underlying file system's ordering is not dependent on file name.

Alterado na versão 3.5: Found packages are now checked for `load_tests` regardless of whether their path matches *pattern*, because it is impossible for a package name to match the default pattern.

The following attributes of a *TestLoader* can be configured either by subclassing or assignment on an instance:

testMethodPrefix
String giving the prefix of method names which will be interpreted as test methods. The default value is `'test'`.

This affects *getTestCaseNames()* and all the `loadTestsFrom*()` methods.

sortTestMethodsUsing
Function to be used to compare method names when sorting them in *getTestCaseNames()* and all the `loadTestsFrom*()` methods.

suiteClass
Callable object that constructs a test suite from a list of tests. No methods on the resulting object are needed. The default value is the *TestSuite* class.

This affects all the `loadTestsFrom*()` methods.

testNamePatterns
List of Unix shell-style wildcard test name patterns that test methods have to match to be included in test suites (see `-v` option).

If this attribute is not `None` (the default), all test methods to be included in test suites must match one of the patterns in this list. Note that matches are always performed using *fnmatch.fnmatchcase()*, so unlike patterns passed to the `-v` option, simple substring patterns will have to be converted using `*` wildcards.

This affects all the `loadTestsFrom*()` methods.

Novo na versão 3.7.

class unittest.**TestResult**
This class is used to compile information about which tests have succeeded and which have failed.

A *TestResult* object stores the results of a set of tests. The *TestCase* and *TestSuite* classes ensure that results are properly recorded; test authors do not need to worry about recording the outcome of tests.

Testing frameworks built on top of *unittest* may want access to the *TestResult* object generated by running a set of tests for reporting purposes; a *TestResult* instance is returned by the `TestRunner.run()` method for this purpose.

TestResult instances have the following attributes that will be of interest when inspecting the results of running a set of tests:

errors
A list containing 2-tuples of *TestCase* instances and strings holding formatted tracebacks. Each tuple represents a test which raised an unexpected exception.

failures
A list containing 2-tuples of *TestCase* instances and strings holding formatted tracebacks. Each tuple represents a test where a failure was explicitly signalled using the `TestCase.assert*()` methods.

skipped
A list containing 2-tuples of *TestCase* instances and strings holding the reason for skipping the test.

Novo na versão 3.1.

expectedFailures
A list containing 2-tuples of *TestCase* instances and strings holding formatted tracebacks. Each tuple represents an expected failure of the test case.

unexpectedSuccesses
A list containing *TestCase* instances that were marked as expected failures, but succeeded.

shouldStop
Set to `True` when the execution of tests should stop by *stop()*.

testsRun
The total number of tests run so far.

buffer
If set to true, `sys.stdout` and `sys.stderr` will be buffered in between *startTest()* and *stopTest()* being called. Collected output will only be echoed onto the real `sys.stdout` and `sys.stderr` if the test fails or errors. Any output is also attached to the failure / error message.

Novo na versão 3.2.

failfast
Se definido como true (verdadeiro) *stop()* será chamado na primeira falha ou erro, interrompendo a execução do teste.

Novo na versão 3.2.

tb_locals
If set to true then local variables will be shown in tracebacks.

Novo na versão 3.5.

wasSuccessful()
Return `True` if all tests run so far have passed, otherwise returns `False`.

Alterado na versão 3.4: Returns `False` if there were any *unexpectedSuccesses* from tests marked with the *expectedFailure()* decorator.

stop()
This method can be called to signal that the set of tests being run should be aborted by setting the

shouldStop attribute to `True`. `TestRunner` objects should respect this flag and return without running any additional tests.

For example, this feature is used by the *TextTestRunner* class to stop the test framework when the user signals an interrupt from the keyboard. Interactive tools which provide `TestRunner` implementations can use this in a similar manner.

The following methods of the *TestResult* class are used to maintain the internal data structures, and may be extended in subclasses to support additional reporting requirements. This is particularly useful in building tools which support interactive reporting while tests are being run.

startTest(*test*)
Called when the test case *test* is about to be run.

stopTest(*test*)
Called after the test case *test* has been executed, regardless of the outcome.

startTestRun()
Called once before any tests are executed.

Novo na versão 3.1.

stopTestRun()
Called once after all tests are executed.

Novo na versão 3.1.

addError(*test*, *err*)
Called when the test case *test* raises an unexpected exception. *err* is a tuple of the form returned by *sys.exc_info()*: `(type, value, traceback)`.

The default implementation appends a tuple `(test, formatted_err)` to the instance's *errors* attribute, where *formatted_err* is a formatted traceback derived from *err*.

addFailure(*test*, *err*)
Called when the test case *test* signals a failure. *err* is a tuple of the form returned by *sys.exc_info()*: `(type, value, traceback)`.

The default implementation appends a tuple `(test, formatted_err)` to the instance's *failures* attribute, where *formatted_err* is a formatted traceback derived from *err*.

addSuccess(*test*)
Called when the test case *test* succeeds.

The default implementation does nothing.

addSkip(*test*, *reason*)
Called when the test case *test* is skipped. *reason* is the reason the test gave for skipping.

The default implementation appends a tuple `(test, reason)` to the instance's *skipped* attribute.

addExpectedFailure(*test*, *err*)
Called when the test case *test* fails, but was marked with the *expectedFailure()* decorator.

The default implementation appends a tuple `(test, formatted_err)` to the instance's *expectedFailures* attribute, where *formatted_err* is a formatted traceback derived from *err*.

addUnexpectedSuccess(*test*)
Called when the test case *test* was marked with the *expectedFailure()* decorator, but succeeded.

The default implementation appends the test to the instance's *unexpectedSuccesses* attribute.

addSubTest(*test*, *subtest*, *outcome*)
Called when a subtest finishes. *test* is the test case corresponding to the test method. *subtest* is a custom `TestCase` instance describing the subtest.

If *outcome* is `None`, the subtest succeeded. Otherwise, it failed with an exception where *outcome* is a tuple of the form returned by `sys.exc_info()`: `(type, value, traceback)`.

The default implementation does nothing when the outcome is a success, and records subtest failures as normal failures.

Novo na versão 3.4.

class unittest.**TextTestResult**(*stream*, *descriptions*, *verbosity*)
A concrete implementation of `TestResult` used by the `TextTestRunner`.

Novo na versão 3.2: This class was previously named `_TextTestResult`. The old name still exists as an alias but is deprecated.

unittest.**defaultTestLoader**
Instance of the `TestLoader` class intended to be shared. If no customization of the `TestLoader` is needed, this instance can be used instead of repeatedly creating new instances.

class unittest.**TextTestRunner**(*stream=None*, *descriptions=True*, *verbosity=1*, *failfast=False*, *buffer=False*, *resultclass=None*, *warnings=None*, ***, *tb_locals=False*)
A basic test runner implementation that outputs results to a stream. If *stream* is `None`, the default, `sys.stderr` is used as the output stream. This class has a few configurable parameters, but is essentially very simple. Graphical applications which run test suites should provide alternate implementations. Such implementations should accept `**kwargs` as the interface to construct runners changes when features are added to unittest.

By default this runner shows `DeprecationWarning`, `PendingDeprecationWarning`, `ResourceWarning` and `ImportWarning` even if they are *ignored by default*. Deprecation warnings caused by *deprecated unittest methods* are also special-cased and, when the warning filters are `'default'` or `'always'`, they will appear only once per-module, in order to avoid too many warning messages. This behavior can be overridden using Python's `-Wd` or `-Wa` options (see Warning control) and leaving *warnings* to `None`.

Alterado na versão 3.2: Added the `warnings` argument.

Alterado na versão 3.2: The default stream is set to `sys.stderr` at instantiation time rather than import time.

Alterado na versão 3.5: Added the tb_locals parameter.

_makeResult()
This method returns the instance of `TestResult` used by `run()`. It is not intended to be called directly, but can be overridden in subclasses to provide a custom `TestResult`.

`_makeResult()` instantiates the class or callable passed in the `TextTestRunner` constructor as the `resultclass` argument. It defaults to `TextTestResult` if no `resultclass` is provided. The result class is instantiated with the following arguments:

```
stream, descriptions, verbosity
```

run(*test*)
This method is the main public interface to the `TextTestRunner`. This method takes a `TestSuite` or `TestCase` instance. A `TestResult` is created by calling `_makeResult()` and the test(s) are run and the results printed to stdout.

unittest.**main**(*module='__main__'*, *defaultTest=None*, *argv=None*, *testRunner=None*, *testLoader=unittest.defaultTestLoader*, *exit=True*, *verbosity=1*, *failfast=None*, *catchbreak=None*, *buffer=None*, *warnings=None*)
A command-line program that loads a set of tests from *module* and runs them; this is primarily for making test modules conveniently executable. The simplest use for this function is to include the following line at the end of a test script:

```
if __name__ == '__main__':
    unittest.main()
```

You can run tests with more detailed information by passing in the verbosity argument:

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

The *defaultTest* argument is either the name of a single test or an iterable of test names to run if no test names are specified via *argv*. If not specified or `None` and no test names are provided via *argv*, all tests found in *module* are run.

O argumento *argv* pode ser uma lista de opções passada para o programa, com o primeiro elemento sendo o nome do programa. Se não for especificado ou for `None`, os valores de *sys.argv* são usados.

The *testRunner* argument can either be a test runner class or an already created instance of it. By default `main` calls *sys.exit()* with an exit code indicating success or failure of the tests run.

The *testLoader* argument has to be a *TestLoader* instance, and defaults to *defaultTestLoader*.

`main` supports being used from the interactive interpreter by passing in the argument `exit=False`. This displays the result on standard output without calling *sys.exit()*:

```
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

The *failfast*, *catchbreak* and *buffer* parameters have the same effect as the same-name *command-line options*.

The *warnings* argument specifies the *warning filter* that should be used while running the tests. If it's not specified, it will remain `None` if a `-W` option is passed to **python** (see Warning control), otherwise it will be set to `'default'`.

Calling `main` actually returns an instance of the `TestProgram` class. This stores the result of the tests run as the `result` attribute.

Alterado na versão 3.1: The *exit* parameter was added.

Alterado na versão 3.2: The *verbosity*, *failfast*, *catchbreak*, *buffer* and *warnings* parameters were added.

Alterado na versão 3.4: The *defaultTest* parameter was changed to also accept an iterable of test names.

load_tests Protocol

Novo na versão 3.2.

Modules or packages can customize how tests are loaded from them during normal test runs or test discovery by implementing a function called `load_tests`.

If a test module defines `load_tests` it will be called by *TestLoader.loadTestsFromModule()* with the following arguments:

```
load_tests(loader, standard_tests, pattern)
```

where *pattern* is passed straight through from `loadTestsFromModule`. It defaults to `None`.

It should return a *TestSuite*.

loader is the instance of *TestLoader* doing the loading. *standard_tests* are the tests that would be loaded by default from the module. It is common for test modules to only want to add or remove tests from the standard set of tests. The third argument is used when loading packages as part of test discovery.

A typical `load_tests` function that loads tests from a specific set of *TestCase* classes may look like:

```
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

If discovery is started in a directory containing a package, either from the command line or by calling *TestLoader.discover()*, then the package `__init__.py` will be checked for `load_tests`. If that function does not exist, discovery will recurse into the package as though it were just another directory. Otherwise, discovery of the package's tests will be left up to `load_tests` which is called with the following arguments:

```
load_tests(loader, standard_tests, pattern)
```

This should return a *TestSuite* representing all the tests from the package. (`standard_tests` will only contain tests collected from `__init__.py`.)

Because the pattern is passed into `load_tests` the package is free to continue (and potentially modify) test discovery. A 'do nothing' `load_tests` function for a test package would look like:

```
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir, pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

Alterado na versão 3.5: Discovery no longer checks package names for matching *pattern* due to the impossibility of package names matching the default pattern.

27.4.9 Classes e Módulos de Definição de Contexto

Definições de contexto em um nível de classe e módulo são implementadas na classe *TestSuite*. Quando a suíte de testes encontrar um teste de uma nova classe, o método `tearDownClass()` da classe anterior (se houver alguma) é chamado logo antes da chamada do método `setUpClass()` da nova classe.

Similarly if a test is from a different module from the previous test then `tearDownModule` from the previous module is run, followed by `setUpModule` from the new module.

Após executar todos os testes, haverá a execução final do `tearDownClass` e do `tearDownModule`.

Note that shared fixtures do not play well with [potential] features like test parallelization and they break test isolation. They should be used with care.

The default ordering of tests created by the unittest test loaders is to group all tests from the same modules and classes together. This will lead to `setUpClass` / `setUpModule` (etc) being called exactly once per class and module. If you randomize the order, so that tests from different modules and classes are adjacent to each other, then these shared fixture functions may be called multiple times in a single test run.

Shared fixtures are not intended to work with suites with non-standard ordering. A `BaseTestSuite` still exists for frameworks that don't want to support shared fixtures.

If there are any exceptions raised during one of the shared fixture functions the test is reported as an error. Because there is no corresponding test instance an `_ErrorHolder` object (that has the same interface as a *TestCase*) is created to

represent the error. If you are just using the standard unittest test runner then this detail doesn't matter, but if you are a framework author it may be relevant.

setUpClass and tearDownClass

These must be implemented as class methods:

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

If you want the `setUpClass` and `tearDownClass` on base classes called then you must call up to them yourself. The implementations in *`TestCase`* are empty.

If an exception is raised during a `setUpClass` then the tests in the class are not run and the `tearDownClass` is not run. Skipped classes will not have `setUpClass` or `tearDownClass` run. If the exception is a *`SkipTest`* exception then the class will be reported as having been skipped instead of as an error.

setUpModule and tearDownModule

These should be implemented as functions:

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

If an exception is raised in a `setUpModule` then none of the tests in the module will be run and the `tearDownModule` will not be run. If the exception is a *`SkipTest`* exception then the module will be reported as having been skipped instead of as an error.

27.4.10 Tratamento de sinal

Novo na versão 3.2.

The *`-c/--catch`* command-line option to unittest, along with the `catchbreak` parameter to *`unittest.main()`*, provide more friendly handling of control-C during a test run. With catch break behavior enabled control-C will allow the currently running test to complete, and the test run will then end and report all the results so far. A second control-c will raise a *`KeyboardInterrupt`* in the usual way.

The control-c handling signal handler attempts to remain compatible with code or tests that install their own *`signal.SIGINT`* handler. If the `unittest` handler is called but *isn't* the installed *`signal.SIGINT`* handler, i.e. it has been replaced by the system under test and delegated to, then it calls the default handler. This will normally be the expected behavior by code that replaces an installed handler and delegates to it. For individual tests that need `unittest` control-c handling disabled the *`removeHandler()`* decorator can be used.

There are a few utility functions for framework authors to enable control-c handling functionality within test frameworks.

unittest.**installHandler**()
: Install the control-c handler. When a *signal.SIGINT* is received (usually in response to the user pressing control-c) all registered results have *stop()* called.

unittest.**registerResult**(*result*)
: Register a *TestResult* object for control-c handling. Registering a result stores a weak reference to it, so it doesn't prevent the result from being garbage collected.

  Registering a *TestResult* object has no side-effects if control-c handling is not enabled, so test frameworks can unconditionally register all results they create independently of whether or not handling is enabled.

unittest.**removeResult**(*result*)
: Remove um resultado registrado. Dado que um resultado for removido, então *stop()* não será mais chamado no objeto resultado em resposta a um Ctrl+C

unittest.**removeHandler**(*function=None*)
: When called without arguments this function removes the control-c handler if it has been installed. This function can also be used as a test decorator to temporarily remove the handler while the test is being executed:

```
@unittest.removeHandler
def test_signal_handling(self):
    ...
```

27.5 unittest.mock — biblioteca de objeto mock

Novo na versão 3.3.

Código Fonte: Lib/unittest/mock.py

---

unittest.mock é uma biblioteca para teste em Python. Que permite substituir partes do seu sistema em teste por objetos simulados e fazer afirmações sobre como elas foram usadas.

: mod:*unittest.mock* fornece uma classe core: class:‘ Mock‘ removendo a necessidade de criar uma série de stubs em todo o seu conjunto de testes. Depois de executar uma ação, você pode fazer afirmações sobre quais métodos / atributos foram usados e com quais argumentos foram chamados. Você também pode especificar valores de retorno e definir os atributos necessários da maneira normal.

Adicionalmente, o mock fornece um decorador *patch()* que lida com os atributos do módulo de patch e do nível de classe no escopo de um teste, junto com *sentinel* para criar objetos únicos. Veja o *guia rápido* para alguns exemplos de como usar :classe:‘ Mock‘,:classe:*MagicMock* e *patch()*.

Mock é muito fácil de usar e foi projetado para uso com *unittest*. O mock é baseado no padrão ‘ação -> asserção’ em vez de ‘gravar -> reproduzir’ usado por muitas estruturas de simulação.

Existe um backport de *unittest.mock* para versões anteriores do Python, disponível como mock no PyPI.

27.5.1 Guia Rápido

Os objetos *Mock* e *MagicMock* criam todos os atributos e métodos à medida que você os acessa e armazena detalhes de como eles foram usados. Você pode configurá-los, especificar valores de retorno ou limitar quais atributos estão disponíveis e, em seguida, fazer afirmações sobre como eles foram usados:

```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

`side_effect` permite que você execute efeitos colaterais, incluindo o lançamento de uma exceção quando um mock é chamado:

```
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
  ...
KeyError: 'foo'
```

```
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

O Mock tem muitas outras maneiras de configurá-lo e controlar seu comportamento. Por exemplo, o argumento *spec* configura o mock para obter sua especificação de outro objeto. Tentar acessar atributos ou métodos no mock que não existem na especificação falhará com um *AttributeError*.

The *patch()* decorator / context manager makes it easy to mock classes or objects in a module under test. The object you specify will be replaced with a mock (or other object) during the test and restored when the test ends:

```
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
...     assert MockClass2 is module.ClassName2
...     assert MockClass1.called
...     assert MockClass2.called
...
>>> test()
```

Nota: Quando você aninha decoradores de patches, as simulações são passadas para a função decorada na mesma ordem em que foram aplicadas (a ordem normal *Python* em que os decoradores são aplicados). Isso significa de baixo para cima, portanto, no exemplo acima, a simulação para `module.ClassName1` é passada primeiro.

Com *patch()*, é importante que você faça o patch de objetos no espaço de nomes onde eles são procurados. Normalmente, isso é simples, mas para um guia rápido, leia *onde fazer o patch*.

Assim como um decorador *patch()* pode ser usado como um gerenciador de contexto em uma instrução with:

```
>>> with patch.object(ProductionClass, 'method', return_value=None) as mock_method:
...     thing = ProductionClass()
...     thing.method(1, 2, 3)
...
>>> mock_method.assert_called_once_with(1, 2, 3)
```

Também existe *patch.dict()* para definir valores em um dicionário apenas durante um escopo e restaurar o dicionário ao seu estado original quando o teste termina:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

Mock possui suporte a simulação de *métodos mágicos* de Python. A maneira mais fácil de usar métodos mágicos é com a classe *MagicMock*. Ele permite que você faça coisas como:

```
>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobarbaz'
>>> str(mock)
'foobarbaz'
>>> mock.__str__.assert_called_with()
```

Mock permite atribuir funções (ou outras instâncias do Mock) a métodos mágicos e elas serão chamadas apropriadamente. A classe *MagicMock* é apenas uma variante do Mock que possui todos os métodos mágicos pré-criados para você (bem, todos os úteis de qualquer maneira).

A seguir, é apresentado um exemplo do uso de métodos mágicos com a classe Mock comum:

```
>>> mock = Mock()
>>> mock.__str__ = Mock(return_value='wheeeeee')
>>> str(mock)
'wheeeeee'
```

For ensuring that the mock objects in your tests have the same api as the objects they are replacing, you can use *auto-speccing*. Auto-speccing can be done through the *autospec* argument to patch, or the *create_autospec()* function. Auto-speccing creates mock objects that have the same attributes and methods as the objects they are replacing, and any functions and methods (including constructors) have the same call signature as the real object.

This ensures that your mocks will fail in the same way as your production code if they are used incorrectly:

```
>>> from unittest.mock import create_autospec
>>> def function(a, b, c):
...     pass
...
>>> mock_function = create_autospec(function, return_value='fishy')
>>> mock_function(1, 2, 3)
'fishy'
>>> mock_function.assert_called_once_with(1, 2, 3)
>>> mock_function('wrong arguments')
```

(continua na próxima página)

(continuação da página anterior)

```
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes exactly 3 arguments (1 given)
```

create_autospec() can also be used on classes, where it copies the signature of the __init__ method, and on callable objects where it copies the signature of the __call__ method.

27.5.2 A classe Mock

Mock is a flexible mock object intended to replace the use of stubs and test doubles throughout your code. Mocks are callable and create attributes as new mocks when you access them[1]. Accessing the same attribute will always return the same mock. Mocks record how you use them, allowing you to make assertions about what your code has done to them.

MagicMock is a subclass of *Mock* with all the magic methods pre-created and ready to use. There are also non-callable variants, useful when you are mocking out objects that aren't callable: *NonCallableMock* and *NonCallableMagicMock*

The *patch()* decorators makes it easy to temporarily replace classes in a particular module with a *Mock* object. By default *patch()* will create a *MagicMock* for you. You can specify an alternative class of *Mock* using the *new_callable* argument to *patch()*.

class unittest.mock.**Mock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

Create a new *Mock* object. *Mock* takes several optional arguments that specify the behaviour of the Mock object:

- *spec*: This can be either a list of strings or an existing object (a class or instance) that acts as the specification for the mock object. If you pass in an object then a list of strings is formed by calling dir on the object (excluding unsupported magic attributes and methods). Accessing any attribute not in this list will raise an *AttributeError*.

  If *spec* is an object (rather than a list of strings) then *__class__* returns the class of the spec object. This allows mocks to pass *isinstance()* tests.

- *spec_set*: A stricter variant of *spec*. If used, attempting to *set* or get an attribute on the mock that isn't on the object passed as *spec_set* will raise an *AttributeError*.

- *side_effect*: A function to be called whenever the Mock is called. See the *side_effect* attribute. Useful for raising exceptions or dynamically changing return values. The function is called with the same arguments as the mock, and unless it returns *DEFAULT*, the return value of this function is used as the return value.

  Alternatively *side_effect* can be an exception class or instance. In this case the exception will be raised when the mock is called.

  If *side_effect* is an iterable then each call to the mock will return the next value from the iterable.

  A *side_effect* can be cleared by setting it to None.

- *return_value*: The value returned when the mock is called. By default this is a new Mock (created on first access). See the *return_value* attribute.

- *unsafe*: By default if any attribute starts with *assert* or *assret* will raise an *AttributeError*. Passing unsafe=True will allow access to these attributes.

  Novo na versão 3.5.

[1] The only exceptions are magic methods and attributes (those that have leading and trailing double underscores). Mock doesn't create these but instead raises an *AttributeError*. This is because the interpreter will often implicitly request these methods, and gets *very* confused to get a new Mock object when it expects a magic method. If you need magic method support see *magic methods*.

- *wraps*: Item for the mock object to wrap. If *wraps* is not `None` then calling the Mock will pass the call through to the wrapped object (returning the real result). Attribute access on the mock will return a Mock object that wraps the corresponding attribute of the wrapped object (so attempting to access an attribute that doesn't exist will raise an *AttributeError*).

  If the mock has an explicit *return_value* set then calls are not passed to the wrapped object and the *return_value* is returned instead.

- *name*: If the mock has a name then it will be used in the repr of the mock. This can be useful for debugging. The name is propagated to child mocks.

Mocks can also be called with arbitrary keyword arguments. These will be used to set attributes on the mock after it is created. See the *configure_mock()* method for details.

assert_called()
Assert that the mock was called at least once.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called()
```

Novo na versão 3.6.

assert_called_once()
Assert that the mock was called exactly once.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
Traceback (most recent call last):
...
AssertionError: Expected 'method' to have been called once. Called 2 times.
```

Novo na versão 3.6.

assert_called_with(**args*, ***kwargs*)
This method is a convenient way of asserting that calls are made in a particular way:

```
>>> mock = Mock()
>>> mock.method(1, 2, 3, test='wow')
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

assert_called_once_with(**args*, ***kwargs*)
Assert that the mock was called exactly once and that that call was with the specified arguments.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
>>> mock('other', bar='values')
>>> mock.assert_called_once_with('other', bar='values')
Traceback (most recent call last):
  ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

assert_any_call(**args*, ***kwargs*)
assert the mock has been called with the specified arguments.

The assert passes if the mock has *ever* been called, unlike *assert_called_with()* and *assert_called_once_with()* that only pass if the call is the most recent one, and in the case of *assert_called_once_with()* it must also be the only call.

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, arg='thing')
>>> mock('some', 'thing', 'else')
>>> mock.assert_any_call(1, 2, arg='thing')
```

assert_has_calls(*calls*, *any_order=False*)
assert the mock has been called with the specified calls. The *mock_calls* list is checked for the calls.

If *any_order* is false then the calls must be sequential. There can be extra calls before or after the specified calls.

If *any_order* is true then the calls can be in any order, but they must all appear in *mock_calls*.

```
>>> mock = Mock(return_value=None)
>>> mock(1)
>>> mock(2)
>>> mock(3)
>>> mock(4)
>>> calls = [call(2), call(3)]
>>> mock.assert_has_calls(calls)
>>> calls = [call(4), call(2), call(3)]
>>> mock.assert_has_calls(calls, any_order=True)
```

assert_not_called()
Assert the mock was never called.

```
>>> m = Mock()
>>> m.hello.assert_not_called()
>>> obj = m.hello()
>>> m.hello.assert_not_called()
Traceback (most recent call last):
  ...
AssertionError: Expected 'hello' to not have been called. Called 1 times.
```

Novo na versão 3.5.

reset_mock(**, return_value=False, side_effect=False*)
The reset_mock method resets all the call attributes on a mock object:

```
>>> mock = Mock(return_value=None)
>>> mock('hello')
>>> mock.called
True
>>> mock.reset_mock()
>>> mock.called
False
```

Alterado na versão 3.6: Added two keyword only argument to the reset_mock function.

This can be useful where you want to make a series of assertions that reuse the same object. Note that *reset_mock() doesn't* clear the return value, *side_effect* or any child attributes you have set using normal assignment by default. In case you want to reset *return_value* or *side_effect*, then pass the corresponding parameter as `True`. Child mocks and the return value mock (if any) are reset as well.

Nota: *return_value*, and *side_effect* are keyword only argument.

mock_add_spec(*spec*, *spec_set=False*)
Add a spec to a mock. *spec* can either be an object or a list of strings. Only attributes on the *spec* can be fetched as attributes from the mock.

If *spec_set* is true then only attributes on the spec can be set.

attach_mock(*mock*, *attribute*)
Attach a mock as an attribute of this one, replacing its name and parent. Calls to the attached mock will be recorded in the *method_calls* and *mock_calls* attributes of this one.

configure_mock(***kwargs*)
Set attributes on the mock through keyword arguments.

Attributes plus return values and side effects can be set on child mocks using standard dot notation and unpacking a dictionary in the method call:

```
>>> mock = Mock()
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock.configure_mock(**attrs)
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

The same thing can be achieved in the constructor call to mocks:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

configure_mock() exists to make it easier to do configuration after the mock has been created.

__dir__()
Mock objects limit the results of `dir(some_mock)` to useful results. For mocks with a *spec* this includes all the permitted attributes for the mock.

See *FILTER_DIR* for what this filtering does, and how to switch it off.

_get_child_mock(***kw*)
Create the child mocks for attributes and return value. By default child mocks will be the same type as the parent. Subclasses of Mock may want to override this to customize the way child mocks are made.

For non-callable mocks the callable variant will be used (rather than any custom subclass).

called
A boolean representing whether or not the mock object has been called:

```
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

call_count

An integer telling you how many times the mock object has been called:

```
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
>>> mock()
>>> mock()
>>> mock.call_count
2
```

return_value

Set this to configure the value returned by calling the mock:

```
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

The default return value is a mock object and you can configure it in the normal way:

```
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
<Mock name='mock()()' id='...'>
>>> mock.return_value.assert_called_with()
```

return_value can also be set in the constructor:

```
>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

side_effect

This can either be a function to be called when the mock is called, an iterable or an exception (class or instance) to be raised.

If you pass in a function it will be called with same arguments as the mock and unless the function returns the *DEFAULT* singleton the call to the mock will then return whatever the function returns. If the function returns *DEFAULT* then the mock will return its normal value (from the *return_value*).

If you pass in an iterable, it is used to retrieve an iterator which must yield a value on every call. This value can either be an exception instance to be raised, or a value to be returned from the call to the mock (*DEFAULT* handling is identical to the function case).

An example of a mock that raises an exception (to test exception handling of an API):

```
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
```

(continua na próxima página)

(continuação da página anterior)

```
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

Using *side_effect* to return a sequence of values:

```
>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)
```

Usando um callable:

```
>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> mock.side_effect = side_effect
>>> mock()
3
```

side_effect can be set in the constructor. Here's an example that adds one to the value the mock is called with and returns it:

```
>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
>>> mock(3)
4
>>> mock(-8)
-7
```

Configuração *side_effect* para `None` limpa isso:

```
>>> m = Mock(side_effect=KeyError, return_value=3)
>>> m()
Traceback (most recent call last):
 ...
KeyError
>>> m.side_effect = None
>>> m()
3
```

call_args

This is either `None` (if the mock hasn't been called), or the arguments that the mock was last called with. This will be in the form of a tuple: the first member is any ordered arguments the mock was called with (or an empty tuple) and the second member is any keyword arguments (or an empty dictionary).

```
>>> mock = Mock(return_value=None)
>>> print(mock.call_args)
None
>>> mock()
>>> mock.call_args
call()
>>> mock.call_args == ()
True
```

(continua na próxima página)

(continuação da página anterior)

```
>>> mock(3, 4)
>>> mock.call_args
call(3, 4)
>>> mock.call_args == ((3, 4),)
True
>>> mock(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args
call(3, 4, 5, key='fish', next='w00t!')
```

call_args, along with members of the lists *call_args_list*, *method_calls* and *mock_calls* are *call* objects. These are tuples, so they can be unpacked to get at the individual arguments and make more complex assertions. See *calls as tuples*.

call_args_list

This is a list of all the calls made to the mock object in sequence (so the length of the list is the number of times it has been called). Before any calls have been made it is an empty list. The *call* object can be used for conveniently constructing lists of calls to compare with *call_args_list*.

```
>>> mock = Mock(return_value=None)
>>> mock()
>>> mock(3, 4)
>>> mock(key='fish', next='w00t!')
>>> mock.call_args_list
[call(), call(3, 4), call(key='fish', next='w00t!')]
>>> expected = [(), ((3, 4),), ({'key': 'fish', 'next': 'w00t!'},)]
>>> mock.call_args_list == expected
True
```

Members of *call_args_list* are *call* objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

method_calls

As well as tracking calls to themselves, mocks also track calls to methods and attributes, and *their* methods and attributes:

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.property.method.attribute()
<Mock name='mock.property.method.attribute()' id='...'>
>>> mock.method_calls
[call.method(), call.property.method.attribute()]
```

Members of *method_calls* are *call* objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

mock_calls

mock_calls records *all* calls to the mock object, its methods, magic methods *and* return value mocks.

```
>>> mock = MagicMock()
>>> result = mock(1, 2, 3)
>>> mock.first(a=3)
<MagicMock name='mock.first()' id='...'>
>>> mock.second()
<MagicMock name='mock.second()' id='...'>
>>> int(mock)
1
```

(continua na próxima página)

(continuação da página anterior)

```
>>> result(1)
<MagicMock name='mock()()' id='...'>
>>> expected = [call(1, 2, 3), call.first(a=3), call.second(),
... call.__int__(), call()(1)]
>>> mock.mock_calls == expected
True
```

Members of *mock_calls* are *call* objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

Nota: The way *mock_calls* are recorded means that where nested calls are made, the parameters of ancestor calls are not recorded and so will always compare equal:

```
>>> mock = MagicMock()
>>> mock.top(a=3).bottom()
<MagicMock name='mock.top().bottom()' id='...'>
>>> mock.mock_calls
[call.top(a=3), call.top().bottom()]
>>> mock.mock_calls[-1] == call.top(a=-1).bottom()
True
```

__class__

Normally the *__class__* attribute of an object will return its type. For a mock object with a `spec`, `__class__` returns the spec class instead. This allows mock objects to pass *isinstance()* tests for the object they are replacing / masquerading as:

```
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

__class__ is assignable to, this allows a mock to pass an *isinstance()* check without forcing you to use a spec:

```
>>> mock = Mock()
>>> mock.__class__ = dict
>>> isinstance(mock, dict)
True
```

class unittest.mock.**NonCallableMock**(*spec=None*, *wraps=None*, *name=None*, *spec_set=None*, ***kwargs*)

A non-callable version of *Mock*. The constructor parameters have the same meaning of *Mock*, with the exception of *return_value* and *side_effect* which have no meaning on a non-callable mock.

Mock objects that use a class or an instance as a `spec` or `spec_set` are able to pass *isinstance()* tests:

```
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
>>> isinstance(mock, SomeClass)
True
```

The *Mock* classes have support for mocking magic methods. See *magic methods* for the full details.

The mock classes and the *patch()* decorators all take arbitrary keyword arguments for configuration. For the *patch()*

decorators the keywords are passed to the constructor of the mock being created. The keyword arguments are for configuring attributes of the mock:

```
>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'
```

The return value and side effect of child mocks can be set in the same way, using dotted notation. As you can't use dotted names directly in a call you have to create a dictionary and unpack it using `**`:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

A callable mock which was created with a *spec* (or a *spec_set*) will introspect the specification object's signature when matching calls to the mock. Therefore, it can match the actual call's arguments regardless of whether they were passed positionally or by name:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, c=3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(1, 2, 3)
>>> mock.assert_called_with(a=1, b=2, c=3)
```

This applies to *assert_called_with()*, *assert_called_once_with()*, *assert_has_calls()* and *assert_any_call()*. When *Autospeccing*, it will also apply to method calls on the mock object.

Alterado na versão 3.4: Added signature introspection on specced and autospecced mock objects.

class unittest.mock.**PropertyMock**(**args*, ***kwargs*)

A mock intended to be used as a property, or other descriptor, on a class. *PropertyMock* provides `__get__()` and `__set__()` methods so you can specify a return value when it is fetched.

Fetching a *PropertyMock* instance from an object calls the mock, with no args. Setting it calls the mock with the value being set.

```
>>> class Foo:
...     @property
...     def foo(self):
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
...
>>> with patch('__main__.Foo.foo', new_callable=PropertyMock) as mock_foo:
...     mock_foo.return_value = 'mockity-mock'
...     this_foo = Foo()
```

(continua na próxima página)

(continuação da página anterior)

```
...     print(this_foo.foo)
...     this_foo.foo = 6
...
mockity-mock
>>> mock_foo.mock_calls
[call(), call(6)]
```

Because of the way mock attributes are stored you can't directly attach a *PropertyMock* to a mock object. Instead you can attach it to the mock type object:

```
>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
3
>>> p.assert_called_once_with()
```

Calling

Mock objects are callable. The call will return the value set as the *return_value* attribute. The default return value is a new Mock object; it is created the first time the return value is accessed (either explicitly or by calling the Mock) - but it is stored and the same one returned each time.

Calls made to the object will be recorded in the attributes like *call_args* and *call_args_list*.

If *side_effect* is set then it will be called after the call has been recorded, so if `side_effect` raises an exception the call is still recorded.

The simplest way to make a mock raise an exception when called is to make *side_effect* an exception class or instance:

```
>>> m = MagicMock(side_effect=IndexError)
>>> m(1, 2, 3)
Traceback (most recent call last):
  ...
IndexError
>>> m.mock_calls
[call(1, 2, 3)]
>>> m.side_effect = KeyError('Bang!')
>>> m('two', 'three', 'four')
Traceback (most recent call last):
  ...
KeyError: 'Bang!'
>>> m.mock_calls
[call(1, 2, 3), call('two', 'three', 'four')]
```

If `side_effect` is a function then whatever that function returns is what calls to the mock return. The `side_effect` function is called with the same arguments as the mock. This allows you to vary the return value of the call dynamically, based on the input:

```
>>> def side_effect(value):
...     return value + 1
...
>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
```

(continua na próxima página)

(continuação da página anterior)

```
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]
```

If you want the mock to still return the default return value (a new mock), or any set return value, then there are two ways of doing this. Either return `mock.return_value` from inside `side_effect`, or return *DEFAULT*:

```
>>> m = MagicMock()
>>> def side_effect(*args, **kwargs):
...     return m.return_value
...
>>> m.side_effect = side_effect
>>> m.return_value = 3
>>> m()
3
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> m.side_effect = side_effect
>>> m()
3
```

To remove a `side_effect`, and return to the default behaviour, set the `side_effect` to `None`:

```
>>> m = MagicMock(return_value=6)
>>> def side_effect(*args, **kwargs):
...     return 3
...
>>> m.side_effect = side_effect
>>> m()
3
>>> m.side_effect = None
>>> m()
6
```

The `side_effect` can also be any iterable object. Repeated calls to the mock will return values from the iterable (until the iterable is exhausted and a *StopIteration* is raised):

```
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
1
>>> m()
2
>>> m()
3
>>> m()
Traceback (most recent call last):
  ...
StopIteration
```

If any members of the iterable are exceptions they will be raised instead of returned:

```
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
```

(continua na próxima página)

(continuação da página anterior)

```
>>> m()
33
>>> m()
Traceback (most recent call last):
  ...
ValueError
>>> m()
66
```

Deleting Attributes

Mock objects create attributes on demand. This allows them to pretend to be objects of any type.

You may want a mock object to return `False` to a *`hasattr()`* call, or raise an *`AttributeError`* when an attribute is fetched. You can do this by providing an object as a `spec` for a mock, but that isn't always convenient.

You "block" attributes by deleting them. Once deleted, accessing an attribute will raise an *`AttributeError`*.

```
>>> mock = MagicMock()
>>> hasattr(mock, 'm')
True
>>> del mock.m
>>> hasattr(mock, 'm')
False
>>> del mock.f
>>> mock.f
Traceback (most recent call last):
    ...
AttributeError: f
```

Mock names and the name attribute

Since "name" is an argument to the *`Mock`* constructor, if you want your mock object to have a "name" attribute you can't just pass it in at creation time. There are two alternatives. One option is to use *`configure_mock()`*:

```
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

A simpler option is to simply set the "name" attribute after mock creation:

```
>>> mock = MagicMock()
>>> mock.name = "foo"
```

Attaching Mocks as Attributes

When you attach a mock as an attribute of another mock (or as the return value) it becomes a "child" of that mock. Calls to the child are recorded in the *method_calls* and *mock_calls* attributes of the parent. This is useful for configuring child mocks and then attaching them to the parent, or for attaching mocks to a parent that records all calls to the children and allows you to make assertions about the order of calls between mocks:

```
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
>>> parent.child2 = child2
>>> child1(1)
>>> child2(2)
>>> parent.mock_calls
[call.child1(1), call.child2(2)]
```

The exception to this is if the mock has a name. This allows you to prevent the "parenting" if for some reason you don't want it to happen.

```
>>> mock = MagicMock()
>>> not_a_child = MagicMock(name='not-a-child')
>>> mock.attribute = not_a_child
>>> mock.attribute()
<MagicMock name='not-a-child()' id='...'>
>>> mock.mock_calls
[]
```

Mocks created for you by *patch()* are automatically given names. To attach mocks that have names to a parent you use the *attach_mock()* method:

```
>>> thing1 = object()
>>> thing2 = object()
>>> parent = MagicMock()
>>> with patch('__main__.thing1', return_value=None) as child1:
...     with patch('__main__.thing2', return_value=None) as child2:
...         parent.attach_mock(child1, 'child1')
...         parent.attach_mock(child2, 'child2')
...         child1('one')
...         child2('two')
...
>>> parent.mock_calls
[call.child1('one'), call.child2('two')]
```

27.5.3 The patchers

The patch decorators are used for patching objects only within the scope of the function they decorate. They automatically handle the unpatching for you, even if exceptions are raised. All of these functions can also be used in with statements or as class decorators.

patch

Nota: *patch()* is straightforward to use. The key is to do the patching in the right namespace. See the section *where to patch*.

unittest.mock.**patch**(*target*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

patch() acts as a function decorator, class decorator or a context manager. Inside the body of the function or with statement, the *target* is patched with a *new* object. When the function/with statement exits the patch is undone.

If *new* is omitted, then the target is replaced with a *MagicMock*. If *patch()* is used as a decorator and *new* is omitted, the created mock is passed in as an extra argument to the decorated function. If *patch()* is used as a context manager the created mock is returned by the context manager.

target should be a string in the form `'package.module.ClassName'`. The *target* is imported and the specified object replaced with the *new* object, so the *target* must be importable from the environment you are calling *patch()* from. The target is imported when the decorated function is executed, not at decoration time.

The *spec* and *spec_set* keyword arguments are passed to the *MagicMock* if patch is creating one for you.

In addition you can pass `spec=True` or `spec_set=True`, which causes patch to pass in the object being mocked as the spec/spec_set object.

new_callable allows you to specify a different class, or callable object, that will be called to create the *new* object. By default *MagicMock* is used.

A more powerful form of *spec* is *autospec*. If you set `autospec=True` then the mock will be created with a spec from the object being replaced. All attributes of the mock will also have the spec of the corresponding attribute of the object being replaced. Methods and functions being mocked will have their arguments checked and will raise a *TypeError* if they are called with the wrong signature. For mocks replacing a class, their return value (the 'instance') will have the same spec as the class. See the *create_autospec()* function and *Autospeccing*.

Instead of `autospec=True` you can pass `autospec=some_object` to use an arbitrary object as the spec instead of the one being replaced.

By default *patch()* will fail to replace attributes that don't exist. If you pass in `create=True`, and the attribute doesn't exist, patch will create the attribute for you when the patched function is called, and delete it again after the patched function has exited. This is useful for writing tests against attributes that your production code creates at runtime. It is off by default because it can be dangerous. With it switched on you can write passing tests against APIs that don't actually exist!

Nota: Alterado na versão 3.5: If you are patching builtins in a module then you don't need to pass `create=True`, it will be added by default.

Patch can be used as a `TestCase` class decorator. It works by decorating each test method in the class. This reduces the boilerplate code when your test methods share a common patchings set. *patch()* finds tests by looking for method names that start with `patch.TEST_PREFIX`. By default this is `'test'`, which matches the way *unittest* finds tests. You can specify an alternative prefix by setting `patch.TEST_PREFIX`.

Patch can be used as a context manager, with the with statement. Here the patching applies to the indented block after the with statement. If you use "as" then the patched object will be bound to the name after the "as"; very useful if *patch()* is creating a mock object for you.

patch() takes arbitrary keyword arguments. These will be passed to the *Mock* (or *new_callable*) on construction.

`patch.dict(...)`, `patch.multiple(...)` and `patch.object(...)` are available for alternate use-cases.

patch() as function decorator, creating the mock for you and passing it into the decorated function:

```
>>> @patch('__main__.SomeClass')
... def function(normal_argument, mock_class):
...     print(mock_class is SomeClass)
...
>>> function(None)
True
```

Patching a class replaces the class with a *MagicMock* *instance*. If the class is instantiated in the code under test then it will be the *return_value* of the mock that will be used.

If the class is instantiated multiple times you could use *side_effect* to return a new mock each time. Alternatively you can set the *return_value* to be anything you want.

To configure return values on methods of *instances* on the patched class you must do this on the `return_value`. For example:

```
>>> class Class:
...     def method(self):
...         pass
...
>>> with patch('__main__.Class') as MockClass:
...     instance = MockClass.return_value
...     instance.method.return_value = 'foo'
...     assert Class() is instance
...     assert Class().method() == 'foo'
...
```

If you use *spec* or *spec_set* and *patch()* is replacing a *class*, then the return value of the created mock will have the same spec.

```
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
>>> assert isinstance(instance, Original)
>>> patcher.stop()
```

The *new_callable* argument is useful where you want to use an alternative class to the default *MagicMock* for the created mock. For example, if you wanted a *NonCallableMock* to be used:

```
>>> thing = object()
>>> with patch('__main__.thing', new_callable=NonCallableMock) as mock_thing:
...     assert thing is mock_thing
...     thing()
...
Traceback (most recent call last):
  ...
TypeError: 'NonCallableMock' object is not callable
```

Another use case might be to replace an object with an *io.StringIO* instance:

```
>>> from io import StringIO
>>> def foo():
...     print('Something')
...
>>> @patch('sys.stdout', new_callable=StringIO)
```

(continua na próxima página)

(continuação da página anterior)

```
... def test(mock_stdout):
...     foo()
...     assert mock_stdout.getvalue() == 'Something\n'
...
>>> test()
```

When *patch()* is creating a mock for you, it is common that the first thing you need to do is to configure the mock. Some of that configuration can be done in the call to patch. Any arbitrary keywords you pass into the call will be used to set attributes on the created mock:

```
>>> patcher = patch('__main__.thing', first='one', second='two')
>>> mock_thing = patcher.start()
>>> mock_thing.first
'one'
>>> mock_thing.second
'two'
```

As well as attributes on the created mock attributes, like the *return_value* and *side_effect*, of child mocks can also be configured. These aren't syntactically valid to pass in directly as keyword arguments, but a dictionary with these as keys can still be expanded into a *patch()* call using **:

```
>>> config = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> patcher = patch('__main__.thing', **config)
>>> mock_thing = patcher.start()
>>> mock_thing.method()
3
>>> mock_thing.other()
Traceback (most recent call last):
  ...
KeyError
```

By default, attempting to patch a function in a module (or a method or an attribute in a class) that does not exist will fail with *AttributeError*:

```
>>> @patch('sys.non_existing_attribute', 42)
... def test():
...     assert sys.non_existing_attribute == 42
...
>>> test()
Traceback (most recent call last):
  ...
AttributeError: <module 'sys' (built-in)> does not have the attribute 'non_existing'
```

but adding `create=True` in the call to *patch()* will make the previous example work as expected:

```
>>> @patch('sys.non_existing_attribute', 42, create=True)
... def test(mock_stdout):
...     assert sys.non_existing_attribute == 42
...
>>> test()
```

patch.object

patch.**object**(*target*, *attribute*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

patch the named member (*attribute*) on an object (*target*) with a mock object.

patch.object() can be used as a decorator, class decorator or a context manager. Arguments *new*, *spec*, *create*, *spec_set*, *autospec* and *new_callable* have the same meaning as for *patch()*. Like *patch()*, *patch.object()* takes arbitrary keyword arguments for configuring the mock object it creates.

When used as a class decorator *patch.object()* honours `patch.TEST_PREFIX` for choosing which methods to wrap.

You can either call *patch.object()* with three arguments or two arguments. The three argument form takes the object to be patched, the attribute name and the object to replace the attribute with.

When calling with the two argument form you omit the replacement object, and a mock is created for you and passed in as an extra argument to the decorated function:

```
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...     SomeClass.class_method(3)
...     mock_method.assert_called_with(3)
...
>>> test()
```

spec, *create* and the other arguments to *patch.object()* have the same meaning as they do for *patch()*.

patch.dict

patch.**dict**(*in_dict*, *values=()*, *clear=False*, ***kwargs*)

Patch a dictionary, or dictionary like object, and restore the dictionary to its original state after the test.

in_dict can be a dictionary or a mapping like container. If it is a mapping then it must at least support getting, setting and deleting items plus iterating over keys.

in_dict can also be a string specifying the name of the dictionary, which will then be fetched by importing it.

values can be a dictionary of values to set in the dictionary. *values* can also be an iterable of `(key, value)` pairs.

If *clear* is true then the dictionary will be cleared before the new values are set.

patch.dict() can also be called with arbitrary keyword arguments to set values in the dictionary.

patch.dict() can be used as a context manager, decorator or class decorator. When used as a class decorator *patch.dict()* honours `patch.TEST_PREFIX` for choosing which methods to wrap.

patch.dict() can be used to add members to a dictionary, or simply let a test change a dictionary, and ensure the dictionary is restored when the test ends.

```
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == {}
```

```
>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
...     print(os.environ['newkey'])
...
newvalue
>>> assert 'newkey' not in os.environ
```

Keywords can be used in the *patch.dict()* call to set values in the dictionary:

```
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule):
...     import mymodule
...     mymodule.function('some', 'args')
...
'fish'
```

patch.dict() can be used with dictionary like objects that aren't actually dictionaries. At the very minimum they must support item getting, setting, deleting and either iteration or membership test. This corresponds to the magic methods `__getitem__()`, `__setitem__()`, `__delitem__()` and either `__iter__()` or `__contains__()`.

```
>>> class Container:
...     def __init__(self):
...         self.values = {}
...     def __getitem__(self, name):
...         return self.values[name]
...     def __setitem__(self, name, value):
...         self.values[name] = value
...     def __delitem__(self, name):
...         del self.values[name]
...     def __iter__(self):
...         return iter(self.values)
...
>>> thing = Container()
>>> thing['one'] = 1
>>> with patch.dict(thing, one=2, two=3):
...     assert thing['one'] == 2
...     assert thing['two'] == 3
...
>>> assert thing['one'] == 1
>>> assert list(thing) == ['one']
```

patch.multiple

patch.**multiple**(*target*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

Perform multiple patches in a single call. It takes the object to be patched (either as an object or a string to fetch the object by importing) and keyword arguments for the patches:

```
with patch.multiple(settings, FIRST_PATCH='one', SECOND_PATCH='two'):
    ...
```

Use *DEFAULT* as the value if you want *patch.multiple()* to create mocks for you. In this case the created mocks are passed into a decorated function by keyword, and a dictionary is returned when *patch.multiple()* is used as a context manager.

patch.multiple() can be used as a decorator, class decorator or a context manager. The arguments *spec*, *spec_set*, *create*, *autospec* and *new_callable* have the same meaning as for *patch()*. These arguments will be applied to *all* patches done by *patch.multiple()*.

When used as a class decorator *patch.multiple()* honours `patch.TEST_PREFIX` for choosing which methods to wrap.

If you want *patch.multiple()* to create mocks for you, then you can use *DEFAULT* as the value. If you use *patch.multiple()* as a decorator then the created mocks are passed into the decorated function by keyword.

```
>>> thing = object()
>>> other = object()
```

```
>>> @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(thing, other):
...     assert isinstance(thing, MagicMock)
...     assert isinstance(other, MagicMock)
...
>>> test_function()
```

patch.multiple() can be nested with other `patch` decorators, but put arguments passed by keyword *after* any of the standard arguments created by *patch()*:

```
>>> @patch('sys.exit')
... @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(mock_exit, other, thing):
...     assert 'other' in repr(other)
...     assert 'thing' in repr(thing)
...     assert 'exit' in repr(mock_exit)
...
>>> test_function()
```

If *patch.multiple()* is used as a context manager, the value returned by the context manager is a dictionary where created mocks are keyed by name:

```
>>> with patch.multiple('__main__', thing=DEFAULT, other=DEFAULT) as values:
...     assert 'other' in repr(values['other'])
...     assert 'thing' in repr(values['thing'])
...     assert values['thing'] is thing
...     assert values['other'] is other
...
```

patch methods: start and stop

All the patchers have `start()` and `stop()` methods. These make it simpler to do patching in `setUp` methods or where you want to do multiple patches without nesting decorators or with statements.

To use them call *patch()*, *patch.object()* or *patch.dict()* as normal and keep a reference to the returned `patcher` object. You can then call `start()` to put the patch in place and `stop()` to undo it.

If you are using *patch()* to create a mock for you then it will be returned by the call to `patcher.start`.

```
>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
>>> assert module.ClassName is not original
```

(continua na próxima página)

(continuação da página anterior)

```
>>> assert module.ClassName is new_mock
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock
```

A typical use case for this might be for doing multiple patches in the `setUp` method of a `TestCase`:

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         self.patcher1 = patch('package.module.Class1')
...         self.patcher2 = patch('package.module.Class2')
...         self.MockClass1 = self.patcher1.start()
...         self.MockClass2 = self.patcher2.start()
...
...     def tearDown(self):
...         self.patcher1.stop()
...         self.patcher2.stop()
...
...     def test_something(self):
...         assert package.module.Class1 is self.MockClass1
...         assert package.module.Class2 is self.MockClass2
...
>>> MyTest('test_something').run()
```

Cuidado: If you use this technique you must ensure that the patching is "undone" by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the `setUp` then `tearDown` is not called. *`unittest.TestCase.addCleanup()`* makes this easier:

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         patcher = patch('package.module.Class')
...         self.MockClass = patcher.start()
...         self.addCleanup(patcher.stop)
...
...     def test_something(self):
...         assert package.module.Class is self.MockClass
...
```

As an added bonus you no longer need to keep a reference to the `patcher` object.

It is also possible to stop all patches which have been started by using *`patch.stopall()`*.

`patch.`**`stopall`**`()`

Stop all active patches. Only stops patches started with `start`.

patch builtins

You can patch any builtins within a module. The following example patches builtin *ord()*:

```
>>> @patch('__main__.ord')
... def test(mock_ord):
...     mock_ord.return_value = 101
...     print(ord('c'))
...
>>> test()
101
```

TEST_PREFIX

All of the patchers can be used as class decorators. When used in this way they wrap every test method on the class. The patchers recognise methods that start with `'test'` as being test methods. This is the same way that the *unittest.TestLoader* finds test methods by default.

It is possible that you want to use a different prefix for your tests. You can inform the patchers of the different prefix by setting `patch.TEST_PREFIX`:

```
>>> patch.TEST_PREFIX = 'foo'
>>> value = 3
>>>
>>> @patch('__main__.value', 'not three')
... class Thing:
...     def foo_one(self):
...         print(value)
...     def foo_two(self):
...         print(value)
...
>>>
>>> Thing().foo_one()
not three
>>> Thing().foo_two()
not three
>>> value
3
```

Nesting Patch Decorators

If you want to perform multiple patches then you can simply stack up the decorators.

You can stack up multiple patch decorators using this pattern:

```
>>> @patch.object(SomeClass, 'class_method')
... @patch.object(SomeClass, 'static_method')
... def test(mock1, mock2):
...     assert SomeClass.static_method is mock1
...     assert SomeClass.class_method is mock2
...     SomeClass.static_method('foo')
...     SomeClass.class_method('bar')
...     return mock1, mock2
...
>>> mock1, mock2 = test()
```

(continua na próxima página)

(continuação da página anterior)

```
>>> mock1.assert_called_once_with('foo')
>>> mock2.assert_called_once_with('bar')
```

Note that the decorators are applied from the bottom upwards. This is the standard way that Python applies decorators. The order of the created mocks passed into your test function matches this order.

Where to patch

patch() works by (temporarily) changing the object that a *name* points to with another one. There can be many names pointing to any individual object, so for patching to work you must ensure that you patch the name used by the system under test.

The basic principle is that you patch where an object is *looked up*, which is not necessarily the same place as where it is defined. A couple of examples will help to clarify this.

Imagine we have a project that we want to test with the following structure:

```
a.py
    -> Defines SomeClass

b.py
    -> from a import SomeClass
    -> some_function instantiates SomeClass
```

Now we want to test `some_function` but we want to mock out `SomeClass` using *patch()*. The problem is that when we import module b, which we will have to do then it imports `SomeClass` from module a. If we use *patch()* to mock out `a.SomeClass` then it will have no effect on our test; module b already has a reference to the *real* `SomeClass` and it looks like our patching had no effect.

The key is to patch out `SomeClass` where it is used (or where it is looked up). In this case `some_function` will actually look up `SomeClass` in module b, where we have imported it. The patching should look like:

```
@patch('b.SomeClass')
```

However, consider the alternative scenario where instead of `from a import SomeClass` module b does `import a` and `some_function` uses `a.SomeClass`. Both of these import forms are common. In this case the class we want to patch is being looked up in the module and so we have to patch `a.SomeClass` instead:

```
@patch('a.SomeClass')
```

Patching Descriptors and Proxy Objects

Both *patch* and *patch.object* correctly patch and restore descriptors: class methods, static methods and properties. You should patch these on the *class* rather than an instance. They also work with *some* objects that proxy attribute access, like the django settings object.

27.5.4 MagicMock and magic method support

Mocking Magic Methods

Mock supports mocking the Python protocol methods, also known as "magic methods". This allows mock objects to replace containers or other objects that implement Python protocols.

Because magic methods are looked up differently from normal methods[2], this support has been specially implemented. This means that only specific magic methods are supported. The supported list includes *almost* all of them. If there are any missing that you need please let us know.

You mock magic methods by setting the method you are interested in to a function or a mock instance. If you are using a function then it *must* take `self` as the first argument[3].

```
>>> def __str__(self):
...     return 'fooble'
...
>>> mock = Mock()
>>> mock.__str__ = __str__
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__str__ = Mock()
>>> mock.__str__.return_value = 'fooble'
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__iter__ = Mock(return_value=iter([]))
>>> list(mock)
[]
```

One use case for this is for mocking objects used as context managers in a `with` statement:

```
>>> mock = Mock()
>>> mock.__enter__ = Mock(return_value='foo')
>>> mock.__exit__ = Mock(return_value=False)
>>> with mock as m:
...     assert m == 'foo'
...
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

Calls to magic methods do not appear in *method_calls*, but they are recorded in *mock_calls*.

Nota: If you use the *spec* keyword argument to create a mock then attempting to set a magic method that isn't in the spec will raise an *AttributeError*.

The full list of supported magic methods is:

- `__hash__`, `__sizeof__`, `__repr__` and `__str__`

[2] Magic methods *should* be looked up on the class rather than the instance. Different versions of Python are inconsistent about applying this rule. The supported protocol methods should work with all supported versions of Python.

[3] The function is basically hooked up to the class, but each `Mock` instance is kept isolated from the others.

- `__dir__`, `__format__` and `__subclasses__`
- `__floor__`, `__trunc__` and `__ceil__`
- Comparisons: `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__` and `__ne__`
- Container methods: `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`, `__len__`, `__iter__`, `__reversed__` and `__missing__`
- Context manager: `__enter__` and `__exit__`
- Unary numeric methods: `__neg__`, `__pos__` and `__invert__`
- The numeric methods (including right hand and in-place variants): `__add__`, `__sub__`, `__mul__`, `__matmul__`, `__div__`, `__truediv__`, `__floordiv__`, `__mod__`, `__divmod__`, `__lshift__`, `__rshift__`, `__and__`, `__xor__`, `__or__`, and `__pow__`
- Numeric conversion methods: `__complex__`, `__int__`, `__float__` and `__index__`
- Descriptor methods: `__get__`, `__set__` and `__delete__`
- Pickling: `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` and `__setstate__`

The following methods exist but are *not* supported as they are either in use by mock, can't be set dynamically, or can cause problems:

- `__getattr__`, `__setattr__`, `__init__` and `__new__`
- `__prepare__`, `__instancecheck__`, `__subclasscheck__`, `__del__`

Magic Mock

There are two `MagicMock` variants: *`MagicMock`* and *`NonCallableMagicMock`*.

class `unittest.mock.`**MagicMock**(**args*, ***kw*)

`MagicMock` is a subclass of *`Mock`* with default implementations of most of the magic methods. You can use `MagicMock` without having to configure the magic methods yourself.

The constructor parameters have the same meaning as for *`Mock`*.

If you use the *spec* or *spec_set* arguments then *only* magic methods that exist in the spec will be created.

class `unittest.mock.`**NonCallableMagicMock**(**args*, ***kw*)

A non-callable version of *`MagicMock`*.

The constructor parameters have the same meaning as for *`MagicMock`*, with the exception of *return_value* and *side_effect* which have no meaning on a non-callable mock.

The magic methods are setup with *`MagicMock`* objects, so you can configure them and use them in the usual way:

```
>>> mock = MagicMock()
>>> mock[3] = 'fish'
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'
>>> mock[2]
'result'
```

By default many of the protocol methods are required to return objects of a specific type. These methods are preconfigured with a default return value, so that they can be used without you having to do anything if you aren't interested in the return value. You can still *set* the return value manually if you want to change the default.

Methods and their defaults:

- `__lt__`: `NotImplemented`
- `__gt__`: `NotImplemented`
- `__le__`: `NotImplemented`
- `__ge__`: `NotImplemented`
- `__int__`: `1`
- `__contains__`: `False`
- `__len__`: `0`
- `__iter__`: `iter([])`
- `__exit__`: `False`
- `__complex__`: `1j`
- `__float__`: `1.0`
- `__bool__`: `True`
- `__index__`: `1`
- `__hash__`: default hash for the mock
- `__str__`: default str for the mock
- `__sizeof__`: default sizeof for the mock

Por exemplo:

```
>>> mock = MagicMock()
>>> int(mock)
1
>>> len(mock)
0
>>> list(mock)
[]
>>> object() in mock
False
```

The two equality methods, `__eq__()` and `__ne__()`, are special. They do the default equality comparison on identity, using the *side_effect* attribute, unless you change their return value to return something else:

```
>>> MagicMock() == 3
False
>>> MagicMock() != 3
True
>>> mock = MagicMock()
>>> mock.__eq__.return_value = True
>>> mock == 3
True
```

The return value of `MagicMock.__iter__()` can be any iterable object and isn't required to be an iterator:

```
>>> mock = MagicMock()
>>> mock.__iter__.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
```

If the return value *is* an iterator, then iterating over it once will consume it and subsequent iterations will result in an empty list:

```
>>> mock.__iter__.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]
```

`MagicMock` has all of the supported magic methods configured except for some of the obscure and obsolete ones. You can still set these up if you want.

Magic methods that are supported but not setup by default in `MagicMock` are:

- `__subclasses__`
- `__dir__`
- `__format__`
- `__get__`, `__set__` and `__delete__`
- `__reversed__` and `__missing__`
- `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` and `__setstate__`
- `__getformat__` and `__setformat__`

27.5.5 Helpers

sentinel

`unittest.mock.`**`sentinel`**

The `sentinel` object provides a convenient way of providing unique objects for your tests.

Attributes are created on demand when you access them by name. Accessing the same attribute will always return the same object. The objects returned have a sensible repr so that test failure messages are readable.

Alterado na versão 3.7: The `sentinel` attributes now preserve their identity when they are *copied* or *pickled*.

Sometimes when testing you need to test that a specific object is passed as an argument to another method, or returned. It can be common to create named sentinel objects to test this. *sentinel* provides a convenient way of creating and testing the identity of objects like this.

In this example we monkey patch `method` to return `sentinel.some_object`:

```
>>> real = ProductionClass()
>>> real.method = Mock(name="method")
>>> real.method.return_value = sentinel.some_object
>>> result = real.method()
>>> assert result is sentinel.some_object
>>> sentinel.some_object
sentinel.some_object
```

DEFAULT

unittest.mock.**DEFAULT**
The *DEFAULT* object is a pre-created sentinel (actually sentinel.DEFAULT). It can be used by *side_effect* functions to indicate that the normal return value should be used.

call

unittest.mock.**call**(**args*, ***kwargs*)
call() is a helper object for making simpler assertions, for comparing with *call_args*, *call_args_list*, *mock_calls* and *method_calls*. *call()* can also be used with *assert_has_calls()*.

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, a='foo', b='bar')
>>> m()
>>> m.call_args_list == [call(1, 2, a='foo', b='bar'), call()]
True
```

call.**call_list**()
For a call object that represents multiple calls, *call_list()* returns a list of all the intermediate calls as well as the final call.

call_list is particularly useful for making assertions on "chained calls". A chained call is multiple calls on a single line of code. This results in multiple entries in *mock_calls* on a mock. Manually constructing the sequence of calls can be tedious.

call_list() can construct the sequence of calls from the same chained call:

```
>>> m = MagicMock()
>>> m(1).method(arg='foo').other('bar')(2.0)
<MagicMock name='mock().method().other()()' id='...'>
>>> kall = call(1).method(arg='foo').other('bar')(2.0)
>>> kall.call_list()
[call(1),
 call().method(arg='foo'),
 call().method().other('bar'),
 call().method().other()(2.0)]
>>> m.mock_calls == kall.call_list()
True
```

A call object is either a tuple of (positional args, keyword args) or (name, positional args, keyword args) depending on how it was constructed. When you construct them yourself this isn't particularly interesting, but the call objects that are in the *Mock.call_args*, *Mock.call_args_list* and *Mock.mock_calls* attributes can be introspected to get at the individual arguments they contain.

The call objects in *Mock.call_args* and *Mock.call_args_list* are two-tuples of (positional args, keyword args) whereas the call objects in *Mock.mock_calls*, along with ones you construct yourself, are three-tuples of (name, positional args, keyword args).

You can use their "tupleness" to pull out the individual arguments for more complex introspection and assertions. The positional arguments are a tuple (an empty tuple if there are no positional arguments) and the keyword arguments are a dictionary:

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, 3, arg='one', arg2='two')
>>> kall = m.call_args
>>> args, kwargs = kall
```

(continua na próxima página)

(continuação da página anterior)

```
>>> args
(1, 2, 3)
>>> kwargs
{'arg2': 'two', 'arg': 'one'}
>>> args is kall[0]
True
>>> kwargs is kall[1]
True
```

```
>>> m = MagicMock()
>>> m.foo(4, 5, 6, arg='two', arg2='three')
<MagicMock name='mock.foo()' id='...'>
>>> kall = m.mock_calls[0]
>>> name, args, kwargs = kall
>>> name
'foo'
>>> args
(4, 5, 6)
>>> kwargs
{'arg2': 'three', 'arg': 'two'}
>>> name is m.mock_calls[0][0]
True
```

create_autospec

unittest.mock.**create_autospec**(*spec*, *spec_set=False*, *instance=False*, ***kwargs*)

Create a mock object using another object as a spec. Attributes on the mock will use the corresponding attribute on the *spec* object as their spec.

Functions or methods being mocked will have their arguments checked to ensure that they are called with the correct signature.

If *spec_set* is `True` then attempting to set attributes that don't exist on the spec object will raise an *AttributeError*.

If a class is used as a spec then the return value of the mock (the instance of the class) will have the same spec. You can use a class as the spec for an instance object by passing `instance=True`. The returned mock will only be callable if instances of the mock are callable.

create_autospec() also takes arbitrary keyword arguments that are passed to the constructor of the created mock.

See *Autospeccing* for examples of how to use auto-speccing with *create_autospec()* and the *autospec* argument to *patch()*.

ANY

unittest.mock.**ANY**

Sometimes you may need to make assertions about *some* of the arguments in a call to mock, but either not care about some of the arguments or want to pull them individually out of *call_args* and make more complex assertions on them.

To ignore certain arguments you can pass in objects that compare equal to *everything*. Calls to *assert_called_with()* and *assert_called_once_with()* will then succeed no matter what was passed in.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
>>> mock.assert_called_once_with('foo', bar=ANY)
```

ANY can also be used in comparisons with call lists like *mock_calls*:

```
>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True
```

FILTER_DIR

unittest.mock.**FILTER_DIR**

FILTER_DIR is a module level variable that controls the way mock objects respond to *dir()* (only for Python 2.6 or more recent). The default is `True`, which uses the filtering described below, to only show useful members. If you dislike this filtering, or need to switch it off for diagnostic purposes, then set `mock.FILTER_DIR = False`.

With filtering on, `dir(some_mock)` shows only useful attributes and will include any dynamically created attributes that wouldn't normally be shown. If the mock was created with a *spec* (or *autospec* of course) then all the attributes from the original are shown, even if they haven't been accessed yet:

```
>>> dir(Mock())
['assert_any_call',
 'assert_called_once_with',
 'assert_called_with',
 'assert_has_calls',
 'attach_mock',
 ...
>>> from urllib import request
>>> dir(Mock(spec=request))
['AbstractBasicAuthHandler',
 'AbstractDigestAuthHandler',
 'AbstractHTTPHandler',
 'BaseHandler',
 ...
```

Many of the not-very-useful (private to *Mock* rather than the thing being mocked) underscore and double underscore prefixed attributes have been filtered from the result of calling *dir()* on a *Mock*. If you dislike this behaviour you can switch it off by setting the module level switch *FILTER_DIR*:

```
>>> from unittest import mock
>>> mock.FILTER_DIR = False
```

(continua na próxima página)

(continuação da página anterior)

```
>>> dir(mock.Mock())
['_NonCallableMock__get_return_value',
 '_NonCallableMock__get_side_effect',
 '_NonCallableMock__return_value_doc',
 '_NonCallableMock__set_return_value',
 '_NonCallableMock__set_side_effect',
 '__call__',
 '__class__',
 ...
```

Alternatively you can just use `vars(my_mock)` (instance members) and `dir(type(my_mock))` (type members) to bypass the filtering irrespective of `mock.FILTER_DIR`.

mock_open

`unittest.mock.`**`mock_open`**(*mock=None*, *read_data=None*)

A helper function to create a mock to replace the use of *open()*. It works for *open()* called directly or used as a context manager.

The *mock* argument is the mock object to configure. If `None` (the default) then a *MagicMock* will be created for you, with the API limited to methods or attributes available on standard file handles.

read_data is a string for the `read()`, *readline()*, and *readlines()* methods of the file handle to return. Calls to those methods will take data from *read_data* until it is depleted. The mock of these methods is pretty simplistic: every time the *mock* is called, the *read_data* is rewound to the start. If you need more control over the data that you are feeding to the tested code you will need to customize this mock for yourself. When that is insufficient, one of the in-memory filesystem packages on PyPI can offer a realistic filesystem for testing.

Alterado na versão 3.4: Added *readline()* and *readlines()* support. The mock of `read()` changed to consume *read_data* rather than returning it on each call.

Alterado na versão 3.5: *read_data* is now reset on each call to the *mock*.

Alterado na versão 3.7.1: Added `__iter__()` to implementation so that iteration (such as in for loops) correctly consumes *read_data*.

Using *open()* as a context manager is a great way to ensure your file handles are closed properly and is becoming common:

```
with open('/some/path', 'w') as f:
    f.write('something')
```

The issue is that even if you mock out the call to *open()* it is the *returned object* that is used as a context manager (and has `__enter__()` and `__exit__()` called).

Mocking context managers with a *MagicMock* is common enough and fiddly enough that a helper function is useful.

```
>>> m = mock_open()
>>> with patch('__main__.open', m):
...     with open('foo', 'w') as h:
...         h.write('some stuff')
...
>>> m.mock_calls
[call('foo', 'w'),
 call().__enter__(),
 call().write('some stuff'),
 call().__exit__(None, None, None)]
```

(continua na próxima página)

(continuação da página anterior)

```
>>> m.assert_called_once_with('foo', 'w')
>>> handle = m()
>>> handle.write.assert_called_once_with('some stuff')
```

And for reading files:

```
>>> with patch('__main__.open', mock_open(read_data='bibble')) as m:
...     with open('foo') as h:
...         result = h.read()
...
>>> m.assert_called_once_with('foo')
>>> assert result == 'bibble'
```

Autospeccing

Autospeccing is based on the existing `spec` feature of mock. It limits the api of mocks to the api of an original object (the spec), but it is recursive (implemented lazily) so that attributes of mocks only have the same api as the attributes of the spec. In addition mocked functions / methods have the same call signature as the original so they raise a *TypeError* if they are called incorrectly.

Before I explain how auto-speccing works, here's why it is needed.

Mock is a very powerful and flexible object, but it suffers from two flaws when used to mock out objects from a system under test. One of these flaws is specific to the *Mock* api and the other is a more general problem with using mock objects.

First the problem specific to *Mock*. *Mock* has two assert methods that are extremely handy: *assert_called_with()* and *assert_called_once_with()*.

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
Traceback (most recent call last):
  ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

Because mocks auto-create attributes on demand, and allow you to call them with arbitrary arguments, if you misspell one of these assert methods then your assertion is gone:

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assret_called_once_with(4, 5, 6)
```

Your tests can pass silently and incorrectly because of the typo.

The second issue is more general to mocking. If you refactor some of your code, rename members and so on, any tests for code that is still using the *old api* but uses mocks instead of the real objects will still pass. This means your tests can all pass even though your code is broken.

Note that this is another reason why you need integration tests as well as unit tests. Testing everything in isolation is all fine and dandy, but if you don't test how your units are "wired together" there is still lots of room for bugs that tests might have caught.

`mock` already provides a feature to help with this, called speccing. If you use a class or instance as the `spec` for a mock then you can only access attributes on the mock that exist on the real class:

```
>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assret_called_with
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
```

The spec only applies to the mock itself, so we still have the same issue with any methods on the mock:

```
>>> mock.has_data()
<mock.Mock object at 0x...>
>>> mock.has_data.assret_called_with()
```

Auto-speccing solves this problem. You can either pass `autospec=True` to *`patch()`* / *`patch.object()`* or use the *`create_autospec()`* function to create a mock with a spec. If you use the `autospec=True` argument to *`patch()`* then the object that is being replaced will be used as the spec object. Because the speccing is done "lazily" (the spec is created as attributes on the mock are accessed) you can use it with very complex or deeply nested objects (like modules that import modules that import modules) without a big performance hit.

Here's an example of it in use:

```
>>> from urllib import request
>>> patcher = patch('__main__.request', autospec=True)
>>> mock_request = patcher.start()
>>> request is mock_request
True
>>> mock_request.Request
<MagicMock name='request.Request' spec='Request' id='...'>
```

You can see that `request.Request` has a spec. `request.Request` takes two arguments in the constructor (one of which is *self*). Here's what happens if we try to call it incorrectly:

```
>>> req = request.Request()
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes at least 2 arguments (1 given)
```

The spec also applies to instantiated classes (i.e. the return value of specced mocks):

```
>>> req = request.Request('foo')
>>> req
<NonCallableMagicMock name='request.Request()' spec='Request' id='...'>
```

`Request` objects are not callable, so the return value of instantiating our mocked out `request.Request` is a non-callable mock. With the spec in place any typos in our asserts will raise the correct error:

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assret_called_with
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
>>> req.add_header.assert_called_with('spam', 'eggs')
```

In many cases you will just be able to add `autospec=True` to your existing *`patch()`* calls and then be protected against bugs due to typos and api changes.

As well as using *autospec* through *`patch()`* there is a *`create_autospec()`* for creating autospecced mocks directly:

```
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request' id='...'>
```

This isn't without caveats and limitations however, which is why it is not the default behaviour. In order to know what attributes are available on the spec object, autospec has to introspect (access attributes) the spec. As you traverse attributes on the mock a corresponding traversal of the original object is happening under the hood. If any of your specced objects have properties or descriptors that can trigger code execution then you may not be able to use autospec. On the other hand it is much better to design your objects so that introspection is safe[4].

A more serious problem is that it is common for instance attributes to be created in the `__init__()` method and not to exist on the class at all. *autospec* can't know about any dynamically created attributes and restricts the api to visible attributes.

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

There are a few different ways of resolving this problem. The easiest, but not necessarily the least annoying, way is to simply set the required attributes on the mock after creation. Just because *autospec* doesn't allow you to fetch attributes that don't exist on the spec it doesn't prevent you setting them:

```
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a = 33
...
```

There is a more aggressive version of both *spec* and *autospec* that *does* prevent you setting non-existent attributes. This is useful if you want to ensure your code only *sets* valid attributes too, but obviously it prevents this particular scenario:

```
>>> with patch('__main__.Something', autospec=True, spec_set=True):
...   thing = Something()
...   thing.a = 33
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

Probably the best way of solving the problem is to add class attributes as default values for instance members initialised in `__init__()`. Note that if you are only setting default attributes in `__init__()` then providing them via class attributes (shared between instances of course) is faster too. e.g.

```
class Something:
    a = 33
```

[4] This only applies to classes or already instantiated objects. Calling a mocked class to create a mock instance *does not* create a real instance. It is only attribute lookups - along with calls to *dir()* - that are done.

This brings up another issue. It is relatively common to provide a default value of `None` for members that will later be an object of a different type. `None` would be useless as a spec because it wouldn't let you access *any* attributes or methods on it. As `None` is *never* going to be useful as a spec, and probably indicates a member that will normally of some other type, autospec doesn't use a spec for members that are set to `None`. These will just be ordinary mocks (well - MagicMocks):

```
>>> class Something:
...     member = None
...
>>> mock = create_autospec(Something)
>>> mock.member.foo.bar.baz()
<MagicMock name='mock.member.foo.bar.baz()' id='...'>
```

If modifying your production classes to add defaults isn't to your liking then there are more options. One of these is simply to use an instance as the spec rather than the class. The other is to create a subclass of the production class and add the defaults to the subclass without affecting the production class. Both of these require you to use an alternative object as the spec. Thankfully *patch()* supports this - you can simply pass the alternative object as the *autospec* argument:

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> class SomethingForTest(Something):
...   a = 33
...
>>> p = patch('__main__.Something', autospec=SomethingForTest)
>>> mock = p.start()
>>> mock.a
<NonCallableMagicMock name='Something.a' spec='int' id='...'>
```

Sealing mocks

unittest.mock.**seal**(*mock*)

Seal will disable the automatic creation of mocks when accessing an attribute of the mock being sealed or any of its attributes that are already mocks recursively.

If a mock instance with a name or a spec is assigned to an attribute it won't be considered in the sealing chain. This allows one to prevent seal from fixing part of the mock object.

```
>>> mock = Mock()
>>> mock.submock.attribute1 = 2
>>> mock.not_submock = mock.Mock(name="sample_name")
>>> seal(mock)
>>> mock.new_attribute  # This will raise AttributeError.
>>> mock.submock.attribute2  # This will raise AttributeError.
>>> mock.not_submock.attribute2  # This won't raise.
```

Novo na versão 3.7.

27.6 `unittest.mock` — começando

Novo na versão 3.3.

27.6.1 Usando Mock

Mock Patching Methods

Usos comuns para a classe *`Mock`* objects incluem:

- Patching methods
- Método de gravação que invoca objetos

You might want to replace a method on an object to check that it is called with the correct arguments by another part of the system:

```
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>
```

Once our mock has been used (`real.method` in this example) it has methods and attributes that allow you to make assertions about how it has been used.

Nota: In most of these examples the *`Mock`* and *`MagicMock`* classes are interchangeable. As the `MagicMock` is the more capable class it makes a sensible one to use by default.

Once the mock has been called its *`called`* attribute is set to `True`. More importantly we can use the *`assert_called_with()`* or *`assert_called_once_with()`* method to check that it was called with the correct arguments.

This example tests that calling `ProductionClass().method` results in a call to the `something` method:

```
>>> class ProductionClass:
...     def method(self):
...         self.something(1, 2, 3)
...     def something(self, a, b, c):
...         pass
...
>>> real = ProductionClass()
>>> real.something = MagicMock()
>>> real.method()
>>> real.something.assert_called_once_with(1, 2, 3)
```

Mock for Method Calls on an Object

In the last example we patched a method directly on an object to check that it was called correctly. Another common use case is to pass an object into a method (or some part of the system under test) and then check that it is used in the correct way.

The simple `ProductionClass` below has a `closer` method. If it is called with an object then it calls `close` on it.

```
>>> class ProductionClass:
...     def closer(self, something):
...         something.close()
...
```

So to test it we need to pass in an object with a `close` method and check that it was called correctly.

```
>>> real = ProductionClass()
>>> mock = Mock()
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```

We don't have to do any work to provide the 'close' method on our mock. Accessing close creates it. So, if 'close' hasn't already been called then accessing it in the test will create it, but *`assert_called_with()`* will raise a failure exception.

Mocking Classes

A common use case is to mock out classes instantiated by your code under test. When you patch a class, then that class is replaced with a mock. Instances are created by *calling the class*. This means you access the "mock instance" by looking at the return value of the mocked class.

In the example below we have a function `some_function` that instantiates `Foo` and calls a method on it. The call to *`patch()`* replaces the class `Foo` with a mock. The `Foo` instance is the result of calling the mock, so it is configured by modifying the mock *`return_value`*.

```
>>> def some_function():
...     instance = module.Foo()
...     return instance.method()
...
>>> with patch('module.Foo') as mock:
...     instance = mock.return_value
...     instance.method.return_value = 'the result'
...     result = some_function()
...     assert result == 'the result'
```

Nomeando os mocks

It can be useful to give your mocks a name. The name is shown in the repr of the mock and can be helpful when the mock appears in test failure messages. The name is also propagated to attributes or methods of the mock:

```
>>> mock = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

Tracking all Calls

Often you want to track more than a single call to a method. The *mock_calls* attribute records all calls to child attributes of the mock - and also to their children.

```
>>> mock = MagicMock()
>>> mock.method()
<MagicMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MagicMock name='mock.attribute.method()' id='...'>
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

If you make an assertion about `mock_calls` and any unexpected methods have been called, then the assertion will fail. This is useful because as well as asserting that the calls you expected have been made, you are also checking that they were made in the right order and with no additional calls:

You use the *call* object to construct lists for comparing with `mock_calls`:

```
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

However, parameters to calls that return mocks are not recorded, which means it is not possible to track nested calls where the parameters used to create ancestors are important:

```
>>> m = Mock()
>>> m.factory(important=True).deliver()
<Mock name='mock.factory().deliver()' id='...'>
>>> m.mock_calls[-1] == call.factory(important=False).deliver()
True
```

Setting Return Values and Attributes

Setting the return values on a mock object is trivially easy:

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

Of course you can do the same for methods on the mock:

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

The return value can also be set in the constructor:

```
>>> mock = Mock(return_value=3)
>>> mock()
3
```

If you need an attribute setting on your mock, just do it:

```
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

Sometimes you want to mock up a more complex situation, like for example `mock.connection.cursor().execute("SELECT 1")`. If we wanted this call to return a list, then we have to configure the result of the nested call.

We can use *call* to construct the set of calls in a "chained call" like this for easy assertion afterwards:

```
>>> mock = Mock()
>>> cursor = mock.connection.cursor.return_value
>>> cursor.execute.return_value = ['foo']
>>> mock.connection.cursor().execute("SELECT 1")
['foo']
>>> expected = call.connection.cursor().execute("SELECT 1").call_list()
>>> mock.mock_calls
[call.connection.cursor(), call.connection.cursor().execute('SELECT 1')]
>>> mock.mock_calls == expected
True
```

It is the call to `.call_list()` that turns our call object into a list of calls representing the chained calls.

Raising exceptions with mocks

A useful attribute is *side_effect*. If you set this to an exception class or instance then the exception will be raised when the mock is called.

```
>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

Side effect functions and iterables

`side_effect` can also be set to a function or an iterable. The use case for `side_effect` as an iterable is where your mock is going to be called several times, and you want each call to return a different value. When you set `side_effect` to an iterable every call to the mock returns the next value from the iterable:

```
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

For more advanced use cases, like dynamically varying the return values depending on what the mock is called with, `side_effect` can be a function. The function will be called with the same arguments as the mock. Whatever the function returns is what the call returns:

```
>>> vals = {(1, 2): 1, (2, 3): 2}
>>> def side_effect(*args):
...     return vals[args]
...
>>> mock = MagicMock(side_effect=side_effect)
>>> mock(1, 2)
1
>>> mock(2, 3)
2
```

Creating a Mock from an Existing Object

One problem with over use of mocking is that it couples your tests to the implementation of your mocks rather than your real code. Suppose you have a class that implements `some_method`. In a test for another class, you provide a mock of this object that *also* provides `some_method`. If later you refactor the first class, so that it no longer has `some_method` - then your tests will continue to pass even though your code is now broken!

Mock allows you to provide an object as a specification for the mock, using the *spec* keyword argument. Accessing methods / attributes on the mock that don't exist on your specification object will immediately raise an attribute error. If you change the implementation of your specification, then tests that use that class will start failing immediately without you having to instantiate the class in those tests.

```
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
Traceback (most recent call last):
   ...
AttributeError: object has no attribute 'old_method'
```

Using a specification also enables a smarter matching of calls made to the mock, regardless of whether some parameters were passed as positional or named arguments:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, 3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(a=1, b=2, c=3)
```

If you want this smarter matching to also work with method calls on the mock, you can use *auto-speccing*.

If you want a stronger form of specification that prevents the setting of arbitrary attributes as well as the getting of them then you can use *spec_set* instead of *spec*.

27.6.2 Patch Decorators

Nota: Com *patch()*, é importante que você faça o patch de objetos no espaço de nomes onde eles são procurados. Normalmente, isso é simples, mas para um guia rápido, leia *onde fazer o patch*.

A common need in tests is to patch a class attribute or a module attribute, for example patching a builtin or patching a class in a module to test that it is instantiated. Modules and classes are effectively global, so patching on them has to be undone after the test or the patch will persist into other tests and cause hard to diagnose problems.

mock provides three convenient decorators for this: *patch()*, *patch.object()* and *patch.dict()*. patch takes a single string, of the form package.module.Class.attribute to specify the attribute you are patching. It also optionally takes a value that you want the attribute (or class or whatever) to be replaced with. 'patch.object' takes an object and the name of the attribute you would like patched, plus optionally the value to patch it with.

patch.object:

```
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
... def test():
...     assert SomeClass.attribute == sentinel.attribute
...
>>> test()
>>> assert SomeClass.attribute == original
```

```
>>> @patch('package.module.attribute', sentinel.attribute)
... def test():
...     from package.module import attribute
...     assert attribute is sentinel.attribute
...
>>> test()
```

If you are patching a module (including *builtins*) then use *patch()* instead of *patch.object()*:

```
>>> mock = MagicMock(return_value=sentinel.file_handle)
>>> with patch('builtins.open', mock):
...     handle = open('filename', 'r')
...
>>> mock.assert_called_with('filename', 'r')
>>> assert handle == sentinel.file_handle, "incorrect file handle returned"
```

The module name can be 'dotted', in the form package.module if needed:

```
>>> @patch('package.module.ClassName.attribute', sentinel.attribute)
... def test():
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
...
>>> test()
```

A nice pattern is to actually decorate test methods themselves:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'attribute', sentinel.attribute)
...     def test_something(self):
...         self.assertEqual(SomeClass.attribute, sentinel.attribute)
...
>>> original = SomeClass.attribute
>>> MyTest('test_something').test_something()
>>> assert SomeClass.attribute == original
```

If you want to patch with a Mock, you can use *patch()* with only one argument (or *patch.object()* with two arguments). The mock will be created for you and passed into the test function / method:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
```

(continua na próxima página)

(continuação da página anterior)

```
...         SomeClass.static_method()
...         mock_method.assert_called_with()
...
>>> MyTest('test_something').test_something()
```

You can stack up multiple patch decorators using this pattern:

```
>>> class MyTest(unittest.TestCase):
...     @patch('package.module.ClassName1')
...     @patch('package.module.ClassName2')
...     def test_something(self, MockClass2, MockClass1):
...         self.assertIs(package.module.ClassName1, MockClass1)
...         self.assertIs(package.module.ClassName2, MockClass2)
...
>>> MyTest('test_something').test_something()
```

When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal *Python* order that decorators are applied). This means from the bottom up, so in the example above the mock for `test_module.ClassName2` is passed in first.

Também existe *`patch.dict()`* para definir valores em um dicionário apenas durante um escopo e restaurar o dicionário ao seu estado original quando o teste termina:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

`patch`, `patch.object` and `patch.dict` can all be used as context managers.

Where you use *`patch()`* to create a mock for you, you can get a reference to the mock using the "as" form of the with statement:

```
>>> class ProductionClass:
...     def method(self):
...         pass
...
>>> with patch.object(ProductionClass, 'method') as mock_method:
...     mock_method.return_value = None
...     real = ProductionClass()
...     real.method(1, 2, 3)
...
>>> mock_method.assert_called_with(1, 2, 3)
```

As an alternative `patch`, `patch.object` and `patch.dict` can be used as class decorators. When used in this way it is the same as applying the decorator individually to every method whose name starts with "test".

27.6.3 Further Examples

Here are some more examples for some slightly more advanced scenarios.

Mocking chained calls

Mocking chained calls is actually straightforward with mock once you understand the *return_value* attribute. When a mock is called for the first time, or you fetch its `return_value` before it has been called, a new *Mock* is created.

This means that you can see how the object returned from a call to a mocked object has been used by interrogating the `return_value` mock:

```
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<Mock name='mock().foo()' id='...'>
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

From here it is a simple step to configure and then make assertions about chained calls. Of course another alternative is writing your code in a more testable way in the first place…

So, suppose we have some code that looks a little bit like this:

```
>>> class Something:
...     def __init__(self):
...         self.backend = BackendProvider()
...     def method(self):
...         response = self.backend.get_endpoint('foobar').create_call('spam', 'eggs
↪').start_call()
...         # more code
```

Assuming that `BackendProvider` is already well tested, how do we test `method()`? Specifically, we want to test that the code section `# more code` uses the response object in the correct way.

As this chain of calls is made from an instance attribute we can monkey patch the `backend` attribute on a `Something` instance. In this particular case we are only interested in the return value from the final call to `start_call` so we don't have much configuration to do. Let's assume the object it returns is 'file-like', so we'll ensure that our response object uses the builtin *open()* as its `spec`.

To do this we create a mock instance as our mock backend and create a mock response object for it. To set the response as the return value for that final `start_call` we could do this:

```
mock_backend.get_endpoint.return_value.create_call.return_value.start_call.return_
↪value = mock_response
```

We can do that in a slightly nicer way using the *configure_mock()* method to directly set the return value for us:

```
>>> something = Something()
>>> mock_response = Mock(spec=open)
>>> mock_backend = Mock()
>>> config = {'get_endpoint.return_value.create_call.return_value.start_call.return_
↪value': mock_response}
>>> mock_backend.configure_mock(**config)
```

With these we monkey patch the "mock backend" in place and can make the real call:

```
>>> something.backend = mock_backend
>>> something.method()
```

Using *mock_calls* we can check the chained call with a single assert. A chained call is several calls in one line of code, so there will be several entries in `mock_calls`. We can use *call.call_list()* to create this list of calls for us:

```
>>> chained = call.get_endpoint('foobar').create_call('spam', 'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

Partial mocking

In some tests I wanted to mock out a call to *datetime.date.today()* to return a known date, but I didn't want to prevent the code under test from creating new date objects. Unfortunately *datetime.date* is written in C, and so I couldn't just monkey-patch out the static `date.today()` method.

I found a simple way of doing this that involved effectively wrapping the date class with a mock, but passing through calls to the constructor to the real class (and returning real instances).

The *patch decorator* is used here to mock out the `date` class in the module under test. The `side_effect` attribute on the mock date class is then set to a lambda function that returns a real date. When the mock date class is called a real date will be constructed and returned by `side_effect`.

```
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
...     mock_date.today.return_value = date(2010, 10, 8)
...     mock_date.side_effect = lambda *args, **kw: date(*args, **kw)
...
...     assert mymodule.date.today() == date(2010, 10, 8)
...     assert mymodule.date(2009, 6, 8) == date(2009, 6, 8)
...
```

Note that we don't patch *datetime.date* globally, we patch `date` in the module that *uses* it. See *where to patch*.

When `date.today()` is called a known date is returned, but calls to the `date(...)` constructor still return normal dates. Without this you can find yourself having to calculate an expected result using exactly the same algorithm as the code under test, which is a classic testing anti-pattern.

Calls to the date constructor are recorded in the `mock_date` attributes (`call_count` and friends) which may also be useful for your tests.

Uma forma alternativa de lidar com datas de mock, ou outras classes embutidas, é discutida nesta entrada de blog.

Mocking a Generator Method

A Python generator is a function or method that uses the `yield` statement to return a series of values when iterated over[1].

A generator method / function is called to return the generator object. It is the generator object that is then iterated over. The protocol method for iteration is *__iter__()*, so we can mock this using a *MagicMock*.

Here's an example class with an "iter" method implemented as a generator:

```
>>> class Foo:
...     def iter(self):
...         for i in [1, 2, 3]:
```

(continua na próxima página)

[1] There are also generator expressions and more advanced uses of generators, but we aren't concerned about them here. A very good introduction to generators and how powerful they are is: Generator Tricks for Systems Programmers.

(continuação da página anterior)

```
...             yield i
...
>>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]
```

How would we mock this class, and in particular its "iter" method?

To configure the values returned from the iteration (implicit in the call to *list*), we need to configure the object returned by the call to `foo.iter()`.

```
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

Applying the same patch to every test method

If you want several patches in place for multiple test methods the obvious way is to apply the patch decorators to every method. This can feel like unnecessary repetition. For Python 2.6 or more recent you can use *patch()* (in all its various forms) as a class decorator. This applies the patches to all test methods on the class. A test method is identified by methods whose names start with `test`:

```
>>> @patch('mymodule.SomeClass')
... class MyTest(TestCase):
...
...     def test_one(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def test_two(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def not_a_test(self):
...         return 'something'
...
>>> MyTest('test_one').test_one()
>>> MyTest('test_two').test_two()
>>> MyTest('test_two').not_a_test()
'something'
```

An alternative way of managing patches is to use the *patch methods: start and stop*. These allow you to move the patching into your `setUp` and `tearDown` methods.

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         self.patcher = patch('mymodule.foo')
...         self.mock_foo = self.patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
...     def tearDown(self):
...         self.patcher.stop()
...
>>> MyTest('test_foo').run()
```

If you use this technique you must ensure that the patching is "undone" by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the setUp then tearDown is not called. *unittest.TestCase.addCleanup()* makes this easier:

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         patcher = patch('mymodule.foo')
...         self.addCleanup(patcher.stop)
...         self.mock_foo = patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
>>> MyTest('test_foo').run()
```

Mocking Unbound Methods

Whilst writing tests today I needed to patch an *unbound method* (patching the method on the class rather than on the instance). I needed self to be passed in as the first argument because I want to make asserts about which objects were calling this particular method. The issue is that you can't patch with a mock for this, because if you replace an unbound method with a mock it doesn't become a bound method when fetched from the instance, and so it doesn't get self passed in. The workaround is to patch the unbound method with a real function instead. The *patch()* decorator makes it so simple to patch out methods with a mock that having to create a real function becomes a nuisance.

If you pass `autospec=True` to patch then it does the patching with a *real* function object. This function object has the same signature as the one it is replacing, but delegates to a mock under the hood. You still get your mock auto-created in exactly the same way as before. What it means though, is that if you use it to patch out an unbound method on a class the mocked function will be turned into a bound method if it is fetched from an instance. It will have `self` passed in as the first argument, which is exactly what I wanted:

```
>>> class Foo:
...   def foo(self):
...     pass
...
>>> with patch.object(Foo, 'foo', autospec=True) as mock_foo:
...   mock_foo.return_value = 'foo'
...   foo = Foo()
...   foo.foo()
...
'foo'
>>> mock_foo.assert_called_once_with(foo)
```

If we don't use `autospec=True` then the unbound method is patched out with a Mock instance instead, and isn't called with `self`.

Checking multiple calls with mock

mock has a nice API for making assertions about how your mock objects are used.

```
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

If your mock is only being called once you can use the `assert_called_once_with()` method that also asserts that the `call_count` is one.

```
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
>>> mock.foo_bar()
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
Traceback (most recent call last):
    ...
AssertionError: Expected to be called once. Called 2 times.
```

Both `assert_called_with` and `assert_called_once_with` make assertions about the *most recent* call. If your mock is going to be called several times, and you want to make assertions about *all* those calls you can use *`call_args_list`*:

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(4, 5, 6)
>>> mock()
>>> mock.call_args_list
[call(1, 2, 3), call(4, 5, 6), call()]
```

The *`call`* helper makes it easy to make assertions about these calls. You can build up a list of expected calls and compare it to `call_args_list`. This looks remarkably similar to the repr of the `call_args_list`:

```
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

Coping with mutable arguments

Another situation is rare, but can bite you, is when your mock is called with mutable arguments. `call_args` and `call_args_list` store *references* to the arguments. If the arguments are mutated by the code under test then you can no longer make assertions about what the values were when the mock was called.

Here's some example code that shows the problem. Imagine the following functions defined in 'mymodule':

```
def frob(val):
    pass

def grob(val):
    "First frob and then clear val"
    frob(val)
    val.clear()
```

When we try to test that `grob` calls `frob` with the correct argument look what happens:

```
>>> with patch('mymodule.frob') as mock_frob:
...     val = {6}
...     mymodule.grob(val)
...
>>> val
set()
>>> mock_frob.assert_called_with({6})
Traceback (most recent call last):
    ...
AssertionError: Expected: (({6},), {})
Called with: ((set(),), {})
```

One possibility would be for mock to copy the arguments you pass in. This could then cause problems if you do assertions that rely on object identity for equality.

Here's one solution that uses the `side_effect` functionality. If you provide a `side_effect` function for a mock then `side_effect` will be called with the same args as the mock. This gives us an opportunity to copy the arguments and store them for later assertions. In this example I'm using *another* mock to store the arguments so that I can use the mock methods for doing the assertion. Again a helper function sets this up for me.

```
>>> from copy import deepcopy
>>> from unittest.mock import Mock, patch, DEFAULT
>>> def copy_call_args(mock):
...     new_mock = Mock()
...     def side_effect(*args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         new_mock(*args, **kwargs)
...         return DEFAULT
...     mock.side_effect = side_effect
...     return new_mock
...
>>> with patch('mymodule.frob') as mock_frob:
...     new_mock = copy_call_args(mock_frob)
...     val = {6}
...     mymodule.grob(val)
...
>>> new_mock.assert_called_with({6})
>>> new_mock.call_args
call({6})
```

`copy_call_args` is called with the mock that will be called. It returns a new mock that we do the assertion on. The `side_effect` function makes a copy of the args and calls our `new_mock` with the copy.

Nota: If your mock is only going to be used once there is an easier way of checking arguments at the point they are called. You can simply do the checking inside a `side_effect` function.

```
>>> def side_effect(arg):
...     assert arg == {6}
...
>>> mock = Mock(side_effect=side_effect)
>>> mock({6})
>>> mock(set())
Traceback (most recent call last):
    ...
AssertionError
```

An alternative approach is to create a subclass of *Mock* or *MagicMock* that copies (using *copy.deepcopy()*) the arguments. Here's an example implementation:

```
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, *args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         return super(CopyingMock, self).__call__(*args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> arg.add(1)
>>> c.assert_called_with(set())
>>> c.assert_called_with(arg)
Traceback (most recent call last):
    ...
AssertionError: Expected call: mock({1})
Actual call: mock(set())
>>> c.foo
<CopyingMock name='mock.foo' id='...'>
```

When you subclass `Mock` or `MagicMock` all dynamically created attributes, and the `return_value` will use your subclass automatically. That means all children of a `CopyingMock` will also have the type `CopyingMock`.

Nesting Patches

Using patch as a context manager is nice, but if you do multiple patches you can end up with nested with statements indenting further and further to the right:

```
>>> class MyTest(TestCase):
...
...     def test_foo(self):
...         with patch('mymodule.Foo') as mock_foo:
...             with patch('mymodule.Bar') as mock_bar:
...                 with patch('mymodule.Spam') as mock_spam:
...                     assert mymodule.Foo is mock_foo
...                     assert mymodule.Bar is mock_bar
...                     assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').test_foo()
>>> assert mymodule.Foo is original
```

With unittest `cleanup` functions and the *patch methods: start and stop* we can achieve the same effect without the nested indentation. A simple helper method, `create_patch`, puts the patch in place and returns the created mock for us:

```
>>> class MyTest(TestCase):
...
...     def create_patch(self, name):
...         patcher = patch(name)
...         thing = patcher.start()
...         self.addCleanup(patcher.stop)
...         return thing
...
```

(continua na próxima página)

(continuação da página anterior)

```
...     def test_foo(self):
...         mock_foo = self.create_patch('mymodule.Foo')
...         mock_bar = self.create_patch('mymodule.Bar')
...         mock_spam = self.create_patch('mymodule.Spam')
...
...         assert mymodule.Foo is mock_foo
...         assert mymodule.Bar is mock_bar
...         assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').run()
>>> assert mymodule.Foo is original
```

Mocking a dictionary with MagicMock

You may want to mock a dictionary, or other container object, recording all access to it whilst having it still behave like a dictionary.

We can do this with *MagicMock*, which will behave like a dictionary, and using *side_effect* to delegate dictionary access to a real underlying dictionary that is under our control.

When the `__getitem__()` and `__setitem__()` methods of our `MagicMock` are called (normal dictionary access) then `side_effect` is called with the key (and in the case of `__setitem__` the value too). We can also control what is returned.

After the `MagicMock` has been used we can use attributes like *call_args_list* to assert about how the dictionary was used:

```
>>> my_dict = {'a': 1, 'b': 2, 'c': 3}
>>> def getitem(name):
...      return my_dict[name]
...
>>> def setitem(name, val):
...     my_dict[name] = val
...
>>> mock = MagicMock()
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

Nota: An alternative to using `MagicMock` is to use `Mock` and *only* provide the magic methods you specifically want:

```
>>> mock = Mock()
>>> mock.__getitem__ = Mock(side_effect=getitem)
>>> mock.__setitem__ = Mock(side_effect=setitem)
```

A *third* option is to use `MagicMock` but passing in `dict` as the *spec* (or *spec_set*) argument so that the `MagicMock` created only has dictionary magic methods available:

```
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

With these side effect functions in place, the `mock` will behave like a normal dictionary but recording the access. It even raises a *KeyError* if you try to access a key that doesn't exist.

```
>>> mock['a']
1
>>> mock['c']
3
>>> mock['d']
Traceback (most recent call last):
    ...
KeyError: 'd'
>>> mock['b'] = 'fish'
>>> mock['d'] = 'eggs'
>>> mock['b']
'fish'
>>> mock['d']
'eggs'
```

After it has been used you can make assertions about the access using the normal mock methods and attributes:

```
>>> mock.__getitem__.call_args_list
[call('a'), call('c'), call('d'), call('b'), call('d')]
>>> mock.__setitem__.call_args_list
[call('b', 'fish'), call('d', 'eggs')]
>>> my_dict
{'a': 1, 'c': 3, 'b': 'fish', 'd': 'eggs'}
```

Mock subclasses and their attributes

There are various reasons why you might want to subclass *Mock*. One reason might be to add helper methods. Here's a silly example:

```
>>> class MyMock(MagicMock):
...     def has_been_called(self):
...         return self.called
...
>>> mymock = MyMock(return_value=None)
>>> mymock
<MyMock id='...'>
>>> mymock.has_been_called()
False
>>> mymock()
>>> mymock.has_been_called()
True
```

The standard behaviour for `Mock` instances is that attributes and the return value mocks are of the same type as the mock they are accessed on. This ensures that `Mock` attributes are `Mocks` and `MagicMock` attributes are `MagicMocks`[2]. So if you're subclassing to add helper methods then they'll also be available on the attributes and return value mock of instances of your subclass.

```
>>> mymock.foo
<MyMock name='mock.foo' id='...'>
>>> mymock.foo.has_been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>
```

(continua na próxima página)

[2] An exception to this rule are the non-callable mocks. Attributes use the callable variant because otherwise non-callable mocks couldn't have callable methods.

(continuação da página anterior)

```
>>> mymock.foo.has_been_called()
True
```

Sometimes this is inconvenient. For example, one user is subclassing mock to created a Twisted adaptor. Having this applied to attributes too actually causes errors.

`Mock` (in all its flavours) uses a method called `_get_child_mock` to create these "sub-mocks" for attributes and return values. You can prevent your subclass being used for attributes by overriding this method. The signature is that it takes arbitrary keyword arguments (`**kwargs`) which are then passed onto the mock constructor:

```
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, **kwargs):
...         return MagicMock(**kwargs)
...
>>> mymock = Subclass()
>>> mymock.foo
<MagicMock name='mock.foo' id='...'>
>>> assert isinstance(mymock, Subclass)
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

Mocking imports with patch.dict

One situation where mocking can be hard is where you have a local import inside a function. These are harder to mock because they aren't using an object from the module namespace that we can patch out.

Generally local imports are to be avoided. They are sometimes done to prevent circular dependencies, for which there is *usually* a much better way to solve the problem (refactor the code) or to prevent "up front costs" by delaying the import. This can also be solved in better ways than an unconditional local import (store the module as a class or module attribute and only do the import on first use).

That aside there is a way to use `mock` to affect the results of an import. Importing fetches an *object* from the *sys.modules* dictionary. Note that it fetches an *object*, which need not be a module. Importing a module for the first time results in a module object being put in *sys.modules*, so usually when you import something you get a module back. This need not be the case however.

This means you can use *patch.dict()* to *temporarily* put a mock in place in *sys.modules*. Any imports whilst this patch is active will fetch the mock. When the patch is complete (the decorated function exits, the with statement body is complete or `patcher.stop()` is called) then whatever was there previously will be restored safely.

Here's an example that mocks out the 'fooble' module.

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     import fooble
...     fooble.blob()
...
<Mock name='mock.blob()' id='...'>
>>> assert 'fooble' not in sys.modules
>>> mock.blob.assert_called_once_with()
```

As you can see the `import fooble` succeeds, but on exit there is no 'fooble' left in *sys.modules*.

This also works for the `from module import name` form:

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     from fooble import blob
...     blob.blip()
...
<Mock name='mock.blob.blip()' id='...'>
>>> mock.blob.blip.assert_called_once_with()
```

With slightly more work you can also mock package imports:

```
>>> mock = Mock()
>>> modules = {'package': mock, 'package.module': mock.module}
>>> with patch.dict('sys.modules', modules):
...     from package.module import fooble
...     fooble()
...
<Mock name='mock.module.fooble()' id='...'>
>>> mock.module.fooble.assert_called_once_with()
```

Tracking order of calls and less verbose call assertions

The *Mock* class allows you to track the *order* of method calls on your mock objects through the *method_calls* attribute. This doesn't allow you to track the order of calls between separate mock objects, however we can use *mock_calls* to achieve the same effect.

Because mocks track calls to child mocks in `mock_calls`, and accessing an arbitrary attribute of a mock creates a child mock, we can create our separate mocks from a parent one. Calls to those child mock will then all be recorded, in order, in the `mock_calls` of the parent:

```
>>> manager = Mock()
>>> mock_foo = manager.foo
>>> mock_bar = manager.bar
```

```
>>> mock_foo.something()
<Mock name='mock.foo.something()' id='...'>
>>> mock_bar.other.thing()
<Mock name='mock.bar.other.thing()' id='...'>
```

```
>>> manager.mock_calls
[call.foo.something(), call.bar.other.thing()]
```

We can then assert about the calls, including the order, by comparing with the `mock_calls` attribute on the manager mock:

```
>>> expected_calls = [call.foo.something(), call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True
```

If `patch` is creating, and putting in place, your mocks then you can attach them to a manager mock using the *attach_mock()* method. After attaching calls will be recorded in `mock_calls` of the manager.

```
>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
```

(continua na próxima página)

(continuação da página anterior)

```
...         manager.attach_mock(MockClass2, 'MockClass2')
...         MockClass1().foo()
...         MockClass2().bar()
...
<MagicMock name='mock.MockClass1().foo()' id='...'>
<MagicMock name='mock.MockClass2().bar()' id='...'>
>>> manager.mock_calls
[call.MockClass1(),
 call.MockClass1().foo(),
 call.MockClass2(),
 call.MockClass2().bar()]
```

If many calls have been made, but you're only interested in a particular sequence of them then an alternative is to use the *assert_has_calls()* method. This takes a list of calls (constructed with the *call* object). If that sequence of calls are in *mock_calls* then the assert succeeds.

```
>>> m = MagicMock()
>>> m().foo().bar().baz()
<MagicMock name='mock().foo().bar().baz()' id='...'>
>>> m.one().two().three()
<MagicMock name='mock.one().two().three()' id='...'>
>>> calls = call.one().two().three().call_list()
>>> m.assert_has_calls(calls)
```

Even though the chained call `m.one().two().three()` aren't the only calls that have been made to the mock, the assert still succeeds.

Sometimes a mock may have several calls made to it, and you are only interested in asserting about *some* of those calls. You may not even care about the order. In this case you can pass `any_order=True` to `assert_has_calls`:

```
>>> m = MagicMock()
>>> m(1), m.two(2, 3), m.seven(7), m.fifty('50')
(...)
>>> calls = [call.fifty('50'), call(1), call.seven(7)]
>>> m.assert_has_calls(calls, any_order=True)
```

More complex argument matching

Using the same basic concept as *ANY* we can implement matchers to do more complex assertions on objects used as arguments to mocks.

Suppose we expect some object to be passed to a mock that by default compares equal based on object identity (which is the Python default for user defined classes). To use *assert_called_with()* we would need to pass in the exact same object. If we are only interested in some of the attributes of this object then we can create a matcher that will check these attributes for us.

You can see in this example how a 'standard' call to `assert_called_with` isn't sufficient:

```
>>> class Foo:
...     def __init__(self, a, b):
...         self.a, self.b = a, b
...
>>> mock = Mock(return_value=None)
>>> mock(Foo(1, 2))
>>> mock.assert_called_with(Foo(1, 2))
```

(continua na próxima página)

(continuação da página anterior)

```
Traceback (most recent call last):
    ...
AssertionError: Expected: call(<__main__.Foo object at 0x...>)
Actual call: call(<__main__.Foo object at 0x...>)
```

A comparison function for our `Foo` class might look something like this:

```
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
...
```

And a matcher object that can use comparison functions like this for its equality operation would look something like this:

```
>>> class Matcher:
...     def __init__(self, compare, some_obj):
...         self.compare = compare
...         self.some_obj = some_obj
...     def __eq__(self, other):
...         return self.compare(self.some_obj, other)
...
```

Putting all this together:

```
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

The `Matcher` is instantiated with our compare function and the `Foo` object we want to compare against. In `assert_called_with` the `Matcher` equality method will be called, which compares the object the mock was called with against the one we created our matcher with. If they match then `assert_called_with` passes, and if they don't an *`AssertionError`* is raised:

```
>>> match_wrong = Matcher(compare, Foo(3, 4))
>>> mock.assert_called_with(match_wrong)
Traceback (most recent call last):
    ...
AssertionError: Expected: ((<Matcher object at 0x...>,), {})
Called with: ((<Foo object at 0x...>,), {})
```

With a bit of tweaking you could have the comparison function raise the *`AssertionError`* directly and provide a more useful failure message.

As of version 1.5, the Python testing library PyHamcrest provides similar functionality, that may be useful here, in the form of its equality matcher (hamcrest.library.integration.match_equality).

27.7 2to3 - Tradução Automatizada de Código Python 2 para 3

2to3 é um programa Python que lê código fonte Python 2.x e aplica uma série de *fixers* para transformá-lo em código válido para a versão do Python 3.x. A biblioteca padrão contém um conjunto rico de *fixers* que lidarão com quase todos os códigos. A biblioteca de suporte *lib2to3* é, no entanto, uma biblioteca genérica e flexível, por isso é possível escrever seus próprios fixers para o 2to3. O módulo *lib2to3* também pode er adaptado a aplicativos personalizados em que o código Python precisa ser editado automaticamente.

27.7.1 Usando o 2to3

O 2to3 geralmente será instalado junto com o interpretador Python como se fosse um script. Ele também está localizado no diretório `Tools/scripts` na raiz da instalação do Python.

Os argumentos básicos de 2to3 são uma lista de arquivos ou diretórios a serem transformados. Os diretórios são recursivamente percorridos pelos fontes Python.

Aqui temos um exemplo de arquivo fonte Python 2.x, `example.py`:

```
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

O mesmo pode ser convertido para código Python 3.x através de 2to3 através da linha de comando:

```
$ 2to3 example.py
```

É impresso um diff contra o arquivo original. O 2to3 também pode escrever as modificações necessárias de volta ao arquivo de origem. (Um backup do arquivo original sempre será feito, salvo se a opção: `-n` for utilizada.) Escrever as alterações de volta está disponível com o uso do flag `-w` flag:

```
$ 2to3 -w example.py
```

Após a transformação, o arquivo `example.py` se parecerá com isso:

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

Os comentários e a indentação são preservados ao longo do processo de tradução.

Por padrão, 2to3 executa um conjunto de *predefined fixers*. A opção de flag `-l` lista todos os fixers disponíveis. Um conjunto explícito de fixers para executação pode ser fornecido com a opção: `-f`. Da mesma forma, a opção `-x` desabilita explicitamente um fixers. O exemplo a seguir executa apenas os fixadores `imports` e `has_key`:

```
$ 2to3 -f imports -f has_key example.py
```

Este comando executa todos os fixers, exceto o fixers `apply`:

```
$ 2to3 -x apply example.py
```

Alguns fixers são *explícitos*, o que significa que eles não são executados por padrão e devem estar listados na linha de comando a para serem executados. Aqui, além dos fixers padrão, o fixers `idioms` também será executado:

```
$ 2to3 -f all -f idioms example.py
```

Observe como a passagem de `all` permite todos os fixers padrão.

Às vezes, 2to3 encontrará um lugar em seu código-fonte que precisa ser alterado, mas o 2to3 não pode corrigir automaticamente. Nesse caso, o 2to3 imprimirá um aviso abaixo do diff para um arquivo. Você deve endereçar o aviso para ter o código 3.x compatível.

O 2to3 também pode refatorar o doctests. Para ativar este modo, use o flag `-d`. Observe que os doctests *somente* serão refatorados. Isso também não exige que o módulo Python seja válido. Por exemplo, os exemplos doctest como em um documento reST também podem ser refatorados com esta opção.

A opção `-v` permite a saída de mais informações sobre o processo de tradução.

Uma vez que algumas declarações de impressão podem ser analisadas como chamadas de função ou declarações, 2to3 nem sempre pode ler arquivos que contêm a função de impressão. Quando 2to3 detecta a presença da diretiva de compilação `from __future__ import print_function`, ele modifica sua gramática interna para interpretar funções *`print()`* como uma função. Esta alteração também pode ser ativada manualmente com o flag `-p`. Use o flag `-p` para executar fixadores no código que já tiveram suas declarações impressas convertidas.

A opção `-o` ou `--output-dir` permite especificar um diretório alternativo para a escrita dos arquivos de saída processados. A flag `-n` é necessária ao usá-lo como arquivos de backup, não faz sentido quando não está sobrescrevendo os arquivos de entrada.

Novo na versão 3.2.3: A opção `-o` foi adicionada.

A flag `-W` ou `--write-unchanged-files`diz ao 2to3 para sempre salvar arquivos de saída, mesmo que nenhuma alteração tenha sido necessária no arquivo. Isso é mais útil com: :option:!-o‘, de modo que uma árvore de código Python inteiro é copiada com a tradução de um diretório para outro. Esta opção implica o uso da flag `-w`, pois não faria sentido de outro modo.

Novo na versão 3.2.3: A opção cujo flag é `-W` foi adicionada.

A opção `--add-suffix` determina a string que será adicionada a todos os nomes de arquivos. O flag da opção `-n` é necessário quando especificamos isso, pois os backups não são necessários quando escrevemos em nomes de arquivos diferentes. Por exemplo:

```
$ 2to3 -n -W --add-suffix=3 example.py
```

Resultará num arquivo convertido de nome `example.py3` a ser escrito.

Novo na versão 3.2.3: A opção `--add-suffix` foi adicionada.

Para traduzir um projeto inteiro de uma árvore de diretório para outra, use:

```
$ 2to3 --output-dir=python3-version/mycode -W -n python2-version/mycode
```

27.7.2 Fixers

Cada passo de transformação do código é encapsulado em um fixer. O comando `2to3 -l` lista todos. Assim como :ref:‘documented above ‘, cada um pode ser ativado ou desativado individualmente. Eles são descritos aqui com mais detalhes.

`apply`
: Remove o uso de `apply()`. Por exemplo `apply(function, *args, **kwargs)` é convertido para `function(*args, **kwargs)`.

`asserts`
: Substitui o nome de método obsoleto *`unittest`* pelo nome correto.

De	Para
`failUnlessEqual(a, b)`	*assertEqual(a, b)*
`assertEquals(a, b)`	*assertEqual(a, b)*
`failIfEqual(a, b)`	*assertNotEqual(a, b)*
`assertNotEquals(a, b)`	*assertNotEqual(a, b)*
`failUnless(a)`	*assertTrue(a)*
`assert_(a)`	*assertTrue(a)*
`failIf(a)`	*assertFalse(a)*
`failUnlessRaises(exc, cal)`	*assertRaises(exc, cal)*
`failUnlessAlmostEqual(a, b)`	*assertAlmostEqual(a, b)*
`assertAlmostEquals(a, b)`	*assertAlmostEqual(a, b)*
`failIfAlmostEqual(a, b)`	*assertNotAlmostEqual(a, b)*
`assertNotAlmostEquals(a, b)`	*assertNotAlmostEqual(a, b)*

basestring
Converte a classe `basestring` para a classe *str*.

buffer
Converte a classe `buffer` para *memoryview*. Este fixers é opcional porque a a API da classe *memoryview* é semelhante, mas não exatamente a mesma que a da classe `buffer`.

dict
Corrige os métodos de iteração de dicionário. `dict.iteritems()` é convertido para *dict.items()*, `dict.iterkeys()` para *dict.keys()*, e `dict.itervalues()` para *dict.values()*. Similarmente temos o método, `dict.viewitems()`, `dict.viewkeys()` e `dict.viewvalues()` que são convertidos respectivamente para *dict.items()*, *dict.keys()* e *dict.values()*. Também encapsula os usos existentes de *dict.items()*, *dict.keys()*, e *dict.values()* em uma chamada para *list*.

except
Converte `except X, T` para `except X as T`.

exec
Converte a `exec` declaração para a função *exec()*.

execfile
Remove o uso da função `execfile()`. O argumento para `execfile()` é encapsulado pelas funções *open()*, *compile()*, e *exec()*.

exitfunc
Mudança de declaração de `sys.exitfunc` para usar o módulo *atexit*.

filter
Encapsula a função *filter()* usando uma chamada para a classe *list*.

funcattrs
Corrige atributos de funções que foram renomeados. Por exemplo, `my_function.func_closure` é convertido para `my_function.__closure__`.

future
Remove a declaração `from __future__ import new_feature`.

getcwdu
Renomeia a função `os.getcwdu()` para *os.getcwd()*.

has_key
Modifica `dict.has_key(key)` para `key in dict`.

idioms
Este fixer opcional executa várias transformações que tornam o código Python mais idiomático. Comparações de tipo como `type(x) is SomeClass` e `type(x) == SomeClass` são convertidas para `isinstance(x, SomeClass)`. `while 1` vira `while True`. Este fixer também tenta usar *sorted()* nos lugares apropriados. Por exemplo, este bloco

```
L = list(some_iterable)
L.sort()
```

é alterado para:

```
L = sorted(some_iterable)
```

import
Detecta importações de irmãos e as converte em importações relativas.

imports
Muda o nome do módulo na biblioteca padrão.

imports2
Lida com outras renomeações de módulos na biblioteca padrão. É separado do fixer *imports* apenas por causa de limitações técnicas.

input
Converte `input(prompt)` para `eval(input(prompt))`.

intern
Converte a função `intern()` para *sys.intern()*.

isinstance
Corrige tipos duplicados no segundo argumento de *isinstance()*. Por exemplo, `isinstance(x, (int, int))` é convertido para `isinstance(x, int)` e `isinstance(x, (int, float, int))` é convertido para `isinstance(x, (int, float))`.

itertools_imports
Remove importações de `itertools.ifilter()`, `itertools.izip()`, e `itertools.imap()`. Importações de `itertools.ifilterfalse()` também são alteradas para *itertools.filterfalse()*.

itertools
Altera o uso de `itertools.ifilter()`, `itertools.izip()`, e `itertools.imap()` para os seus equivalentes incorporados. `itertools.ifilterfalse()` é alterado para *itertools.filterfalse()*.

long
Renomeia a classe `long` para *int*.

map
Encapsula a função *map()* numa chamada a classe *list*. Isso também altera `map(None, x)` para `list(x)`. Usando `from future_builtins import map` desabilitará esse fixers.

metaclass
Converte a sintaxe da metaclasse antiga (`__metaclass__ = Meta` in the class body) para o novo formato (`class X(metaclass=Meta)`).

methodattrs
Corrige nomes de atributos de métodos antigos. Por exemplo `meth.im_func` é convertido para `meth.__func__`.

ne
Converte a sintaxe antiga "diferente", <>, para `!=`.

next
Converte o uso de métodos de iterador `next()` para a função *next()*. Também renomeia métodos *next()* para *__next__()*.

nonzero
Renomeia o méotdo `__nonzero__()` para `__bool__()`.

numliterals
Converte os literais octal para a nova sintaxe.

operator
Converte chamadas para várias funções no módulo *operator* para chamadas de função diferentes, mas equivalentes. Quando necessário, são adicionadas as declarações `import` adequadas, por exemplo `import collections.abc`. Os seguintes mapeamento são feitos:

De	Para
`operator.isCallable(obj)`	`chamável(obj)`
`operator.sequenceIncludes(obj)`	`operator.contains(obj)`
`operator.isSequenceType(obj)`	`isinstance(obj, collections.abc.Sequence)`
`operator.isMappingType(obj)`	`isinstance(obj, collections.abc.Mapping)`
`operator.isNumberType(obj)`	`isinstance(obj, numbers.Number)`
`operator.repeat(obj, n)`	`operator.mul(obj, n)`
`operator.irepeat(obj, n)`	`operator.imul(obj, n)`

paren
Adiciona parênteses onde os mesmos não eram necessários em lista comprehensions. Por exemplo, `[x for x in 1, 2]` becomes `[x for x in (1, 2)]`.

print
Converte a declaração `print` para a função *print()*.

raise
Converte `raise E, V` para `raise E(V)`, e `raise E, V, T` para `raise E(V).with_traceback(T)`. Se `E` for uma tupla, a tradução ficará incorreta porque a substituição de tuplas por exceções foi removida no Python 3x.

raw_input
Converte a função `raw_input()` para *input()*.

reduce
Manipula o movimento de `reduce()` para *functools.reduce()*.

reload
Converte a função `reload()` para `importlib:reload()`.

renames
Altera o `sys.maxint` para *sys.maxsize*.

repr
Substitui o backtick repr pela função *repr()*.

set_literal
Substitui o uso da classe *set* construtor pelo seu literal. Este fixers é opcional.

standarderror
Renomeia `StandardError` para *Exception*.

sys_exc
Altera o obsoleto `sys.exc_value`, `sys.exc_type`, `sys.exc_traceback` para utilizar agora a função *sys.exc_info()*.

throw
Corrige a mudança de API no método gerador `throw()`.

tuple_params
Remove o desempacotamento implícito do parâmetro da tupla. Este fixers insere variáveis temporárias.

types
Corrige o código quebrado pela remoção de alguns membros no módulo *types*.

unicode
Renomeia a classe `unicode` para *str*.

urllib
Manipula a renomeação dos módulos *urllib* e `urllib2` para o pacote *urllib*.

ws_comma
Remove o espaço excessivo de itens separados por vírgulas. Este fixers é opcional.

xrange
Renomeia a função `xrange()` para *range()* e encapsula a chamada para função existente *range()* com *list*.

xreadlines
Altera de `for x in file.xreadlines()` para `for x in file`.

zip
Encapsula o uso da função *zip()* na chamada a classe *list*. Isso está desativado quando `from future_builtins import zip` aparecer.

27.7.3 `lib2to3` - biblioteca 2to3's

Código Fonte: Lib/lib2to3/

Nota: A API do módulo *lib2to3* deve ser considerado instável e pode mudar drasticamente no futuro.

27.8 `test` — Pacote de Testes de Regressão do Python

Nota: O pacote *test* é apenas para uso interno do Python. O mesmo está sendo documentado para o benefício dos principais desenvolvedores do Python. Qualquer uso deste pacote fora da biblioteca padrão do Python é desencorajado, pois, o código mencionado aqui pode ser alterado ou removido sem aviso prévio entre as versões do Python.

O pacote *test* contém todos os testes de regressão do Python, bem como os módulos *test.support* e `test.regrtest`. O *test.support* é usado para aprimorar seus testes enquanto o :mod:‘ test.regrtest‘ conduz (guia) o conjunto de testes.

Cada módulo no pacote :mod: *test* cujo nome começa com `test_` é um conjunto de testes para um módulo ou recurso específico. Todos os novos testes devem ser escritos usando o módulo *unittest* ou *doctest*. Alguns testes mais

antigos são escritos usando um estilo de teste "tradicional" que compara a saída impressa em `sys.stdout`; Este estilo de teste foi considerado obsoleto.

Ver também:

Módulo *unittest* Escrevendo testes de regressão PyUnit.

Módulo *doctest* Testes embutidos em Strings da documentação.

27.8.1 Escrever testes unitáriops para o pacote `test`

É preferível que os testes que usam o módulo *unittest* sigam algumas diretrizes. Uma é nomear o módulo de teste iniciando-o com `test_` e termine com o nome do módulo que está sendo testado. Os métodos de teste no módulo de teste deve começar com `test_` e terminar com uma descrição do que o método está testando. Isso é necessário para que os métodos sejam reconhecidos pelo driver de teste como métodos de teste. Além disso, na string de documentação para o método deve ser incluído. Um comentário (como os `# Tests function returns only True or False`) deve ser usado para fornecer documentação para testar métodos. Isso é feito porque as strings de documentação são impressas se existem e, portanto, qual teste está sendo executado não é indicado.

Um boilerplate básico é muitas vezes usado:

```
import unittest
from test import support

class MyTestCase1(unittest.TestCase):

    # Only use setUp() and tearDown() if necessary

    def setUp(self):
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...

    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

... more test classes ...

if __name__ == '__main__':
    unittest.main()
```

Este padrão de código permite que o conjunto de testes seja executado pelo `test.regrtest`, por conta própria, como um script que suporte o *unittest* CLI, ou através do `python -m unittest` CLI.

O objetivo do teste de regressão é tentar quebrar o código. Isso leva a algumas diretrizes que devemos seguir:

- O conjunto de testes deve exercitar todas as classes, funções e constantes. Isso inclui não apenas a API externa que deve ser apresentada ao mundo exterior, mas também o código "privado".

- Os testes de Whitebox (que examinam o código que está sendo testado quando os testes estão sendo gravados) são preferidos. O teste Blackbox (que testa apenas a interface do público de usuário) não é completo o suficiente para garantir que todos os casos de limite e extremos sejam testados.
- Certifique-se de que todos os valores possíveis sejam testados, incluindo os inválidos. Isso garante que não apenas todos os valores válidos são aceitos, mas também, que os valores impróprios são tratados corretamente.
- Esgote o maior número possível de caminhos de código. Teste onde ocorre a ramificação e, assim, personalize a entrada para garantir que tantos caminhos diferentes pelo código sejam tomados.
- Adicione um teste explícito para quaisquer bugs descobertos ao código testado. Isso garantirá que o erro não apareça novamente se o código for alterado no futuro.
- Certifique-se de limpar após seus testes (como fechar e remover todos os arquivos temporários).
- Se um teste depende de uma condição específica do sistema operacional, então verifique se a condição já existe antes de tentar o teste.
- Importe o menor número de módulos possível e faça isso o mais rápido possível. Isso minimiza dependências externas de testes, e também minimiza possíveis comportamento anômalo dos efeitos colaterais da importação de um módulo.
- Tente maximizar a reutilização de código. Ocasionalmente, os testes variam em algo tão pequeno quanto o tipo de entrada é usado. Minimize a duplicação de código subclassificando uma classe de teste básica com uma classe que especifica o input:

```
class TestFuncAcceptsSequencesMixin:

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = [1, 2, 3]

class AcceptStrings(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = (1, 2, 3)
```

O pacote *test* pode ser executado como um script para conduzir a regressão do Python conjunto de testes, graças à opção -m: **python -m test**. Nos bastidores, ele usa `test.regrtest`; a chamada **python -m test.regrtest** usado nas versões anteriores do Python ainda funciona. Executando o script por si só começa a executar todos os testes de regressão no pacote `test. Ele faz isso encontrando todos os módulos no pacote cujo nome começa com ``test_`, importando-os e executando a função `test_main()` se presente ou carregando os testes via unittest.TestLoader.loadTestsFromModule se `test_main` não existir. Os nomes dos testes a serem executados também podem ser passados para o script. Especificando um teste de regressão simples (**python -m test test_spam**) minimizará saiba e imprima apenas se o teste passou ou falhou.

Ver também:

Test Driven Development Um livro de Kent Beck sobre escrita de testes antes do código.

27.8.2 Executando testes usando a interface de linha de comando

O pacote *test* pode ser executado como um script para conduzir a regressão do Python conjunto de testes, graças à opção -m: **python -m test**. Nos bastidores, ele usa `test.regrtest`; a chamada **python -m test.regrtest** usado nas versões anteriores do Python ainda funciona. Executando o script por si só começa a executar todos os testes de regressão no pacote `test. Ele faz isso encontrando todos os módulos no pacote cujo nome começa com ``test_`, importando-os e executando a função `test_main()` se presente ou carregando os testes via unittest.TestLoader.loadTestsFromModule se `test_main` não existir. Os nomes dos testes a serem executados também podem ser passados para o script. Especificando um teste de regressão simples (**python -m test test_spam**) minimizará saiba e imprima apenas se o teste passou ou falhou.

Running *test* directly allows what resources are available for tests to use to be set. You do this by using the `-u` command-line option. Specifying `all` as the value for the `-u` option enables all possible resources: **python -m test -uall**. If all but one resource is desired (a more common case), a comma-separated list of resources that are not desired may be listed after `all`. The command **python -m test -uall,-audio,-largefile** will run *test* with all resources except the `audio` and `largefile` resources. For a list of all resources and more command-line options, run **python -m test -h**.

Alguns outros meios para executar os testes de regressão dependem em qual plataforma os testes estão sendo executados. No Unix, você pode executar: programa: ' make test' no diretório de mais alto nível onde o Python foi construído. No Windows, executar: programa 'rt.bat' do seu diretório: file: 'PCbuild' executará todos os testes de regressão.

27.9 `test.support` — Utilitários para o conjunto de teste do Python

O: mod: módulo 'test suport' fornece suporte para a suite de teste de regressão do Python.

Nota: *test.support* is not a public module. It is documented here to help Python developers write tests. The API of this module is subject to change without backwards compatibility concerns between releases.

Esse módulo define as seguintes exceções:

exception `test.support.`**TestFailed**
: Exception to be raised when a test fails. This is deprecated in favor of *unittest*-based tests and *unittest.TestCase*'s assertion methods.

exception `test.support.`**ResourceDenied**
: Subclass of *unittest.SkipTest*. Raised when a resource (such as a network connection) is not available. Raised by the *requires()* function.

O módulo :mod:'test.support' define as seguintes constantes:

`test.support.`**verbose**
: `True` when verbose output is enabled. Should be checked when more detailed information is desired about a running test. *verbose* is set by `test.regrtest`.

`test.support.`**is_jython**
: `True` se o interpretador em execução for Jython.

`test.support.`**is_android**
: `True` se o sistema é Android.

`test.support.`**unix_shell**
: Caminho para o console se não estiver no Windows; por outro lado `None`

`test.support.`**FS_NONASCII**
: A non-ASCII character encodable by *os.fsencode()*.

test.support.**TESTFN**
Set to a name that is safe to use as the name of a temporary file. Any temporary file that is created should be closed and unlinked (removed).

test.support.**TESTFN_UNICODE**
Define um nome não-ASCII para o arquivo temporário.

test.support.**TESTFN_ENCODING**
Set to *sys.getfilesystemencoding()*.

test.support.**TESTFN_UNENCODABLE**
Define o nome de arquivo (tipo str) que não pode ser codificado pela codificação do sistema de arquivos no modo estrito. Ele pode ser `None` se não for possível gerar como um nome de arquivo.

test.support.**TESTFN_UNDECODABLE**
Define o nome de arquivo (tipo str) que não pode ser codificado pela codificação do sistema de arquivos no modo estrito. Ele pode ser `None` se não for possível ser gerado com um nome de arquivo.

test.support.**TESTFN_NONASCII**
Set to a filename containing the *FS_NONASCII* character.

test.support.**IPV6_ENABLED**
Set to `True` if IPV6 is enabled on this host, `False` otherwise.

test.support.**SAVEDCWD**
Set to *os.getcwd()*.

test.support.**PGO**
Set when tests can be skipped when they are not useful for PGO.

test.support.**PIPE_MAX_SIZE**
A constant that is likely larger than the underlying OS pipe buffer size, to make writes blocking.

test.support.**SOCK_MAX_SIZE**
A constant that is likely larger than the underlying OS socket buffer size, to make writes blocking.

test.support.**TEST_SUPPORT_DIR**
Define o diretório de mais alto nível que contém *test.support*.

test.support.**TEST_HOME_DIR**
Define o diretório de mais alto nível para o pacote de teste.

test.support.**TEST_DATA_DIR**
Set to the `data` directory within the test package.

test.support.**MAX_Py_ssize_t**
Define :data:‘sys.maxsize‘para grandes testes de memória.

test.support.**max_memuse**
Set by *set_memlimit()* as the memory limit for big memory tests. Limited by *MAX_Py_ssize_t*.

test.support.**real_max_memuse**
Set by *set_memlimit()* as the memory limit for big memory tests. Not limited by *MAX_Py_ssize_t*.

test.support.**MISSING_C_DOCSTRINGS**
Return `True` if running on CPython, not on Windows, and configuration not set with `WITH_DOC_STRINGS`.

test.support.**HAVE_DOCSTRINGS**
Check for presence of docstrings.

test.support.**TEST_HTTP_URL**
Define the URL of a dedicated HTTP server for the network tests.

test.support.**ALWAYS_EQ**
Object that is equal to anything. Used to test mixed type comparison.

test.support.**LARGEST**
Object that is greater than anything (except itself). Used to test mixed type comparison.

test.support.**SMALLEST**
Object that is less than anything (except itself). Used to test mixed type comparison.

O módulo *test.support* define as seguintes funções:

test.support.**forget**(*module_name*)
Remove the module named *module_name* from sys.modules and delete any byte-compiled files of the module.

test.support.**unload**(*name*)
Exclui o *name* de sys.modules.

test.support.**unlink**(*filename*)
Call *os.unlink()* on *filename*. On Windows platforms, this is wrapped with a wait loop that checks for the existence fo the file.

test.support.**rmdir**(*filename*)
Call *os.rmdir()* on *filename*. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the file.

test.support.**rmtree**(*path*)
Call *shutil.rmtree()* on *path* or call *os.lstat()* and *os.rmdir()* to remove a path and its contents. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the files.

test.support.**make_legacy_pyc**(*source*)
Move a PEP 3147/488 pyc file to its legacy pyc location and return the file system path to the legacy pyc file. The *source* value is the file system path to the source file. It does not need to exist, however the PEP 3147/488 pyc file must exist.

test.support.**is_resource_enabled**(*resource*)
Return True if *resource* is enabled and available. The list of available resources is only set when test.regrtest is executing the tests.

test.support.**python_is_optimized**()
Return True if Python was not built with -O0 or -Og.

test.support.**with_pymalloc**()
Retorna _testcapi.WITH_PYMALLOC.

test.support.**requires**(*resource*, *msg=None*)
Raise *ResourceDenied* if *resource* is not available. *msg* is the argument to *ResourceDenied* if it is raised. Always returns True if called by a function whose __name__ is '__main__'. Used when tests are executed by test.regrtest.

test.support.**system_must_validate_cert**(*f*)
Raise *unittest.SkipTest* on TLS certification validation failures.

test.support.**sortdict**(*dict*)
Return a repr of *dict* with keys sorted.

test.support.**findfile**(*filename*, *subdir=None*)
Return the path to the file named *filename*. If no match is found *filename* is returned. This does not equal a failure since it could be the path to the file.

Setting *subdir* indicates a relative path to use to find the file rather than looking directly in the path directories.

test.support.**create_empty_file**(*filename*)
Create an empty file with *filename*. If it already exists, truncate it.

test.support.**fd_count**()
Conte o número de descritores de arquivos abertos.

test.support.**match_test**(*test*)
Match *test* to patterns set in *set_match_tests()*.

test.support.**set_match_tests**(*patterns*)
Define match test with regular expression *patterns*.

test.support.**run_unittest**(**classes*)
Execute *unittest.TestCase* subclasses passed to the function. The function scans the classes for methods starting with the prefix `test_` and executes the tests individually.

It is also legal to pass strings as parameters; these should be keys in `sys.modules`. Each associated module will be scanned by `unittest.TestLoader.loadTestsFromModule()`. This is usually seen in the following `test_main()` function:

```
def test_main():
    support.run_unittest(__name__)
```

Isso executará todos os testes definidos no modulo nomeado.

test.support.**run_doctest**(*module*, *verbosity=None*, *optionflags=0*)
Run *doctest.testmod()* on the given *module*. Return `(failure_count, test_count)`.

If *verbosity* is `None`, *doctest.testmod()* is run with verbosity set to *verbose*. Otherwise, it is run with verbosity set to `None`. *optionflags* is passed as `optionflags` to *doctest.testmod()*.

test.support.**setswitchinterval**(*interval*)
Set the *sys.setswitchinterval()* to the given *interval*. Defines a minimum interval for Android systems to prevent the system from hanging.

test.support.**check_impl_detail**(***guards*)
Use this check to guard CPython's implementation-specific tests or to run them only on the implementations guarded by the arguments:

```
check_impl_detail()               # Only on CPython (default).
check_impl_detail(jython=True)    # Only on Jython.
check_impl_detail(cpython=False)  # Everywhere except CPython.
```

test.support.**check_warnings**(**filters*, *quiet=True*)
A convenience wrapper for *warnings.catch_warnings()* that makes it easier to test that a warning was correctly raised. It is approximately equivalent to calling `warnings.catch_warnings(record=True)` with *warnings.simplefilter()* set to `always` and with the option to automatically validate the results that are recorded.

`check_warnings` accepts 2-tuples of the form `("message regexp", WarningCategory)` as positional arguments. If one or more *filters* are provided, or if the optional keyword argument *quiet* is `False`, it checks to make sure the warnings are as expected: each specified filter must match at least one of the warnings raised by the enclosed code or the test fails, and if any warnings are raised that do not match any of the specified filters the test fails. To disable the first of these checks, set *quiet* to `True`.

Se nenhum argumento é especificado, o padrão é:

```
check_warnings(("", Warning), quiet=True)
```

Nesse caso, todos os avisos são capturados e nenhum erro é gerado.

On entry to the context manager, a `WarningRecorder` instance is returned. The underlying warnings list from *catch_warnings()* is available via the recorder object's *warnings* attribute. As a convenience, the attributes of the object representing the most recent warning can also be accessed directly through the recorder object (see

example below). If no warning has been raised, then any of the attributes that would otherwise be expected on an object representing a warning will return `None`.

The recorder object also has a `reset()` method, which clears the warnings list.

O gerenciador de contexto é desenhado para ser utilizado dessa forma:

```
with check_warnings(("assertion is always true", SyntaxWarning),
                    ("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

No caso, se um aviso não foi criado, ou algum outro aviso não foi criado, :func:‘ *check_warnings* deveria aparecer como um erro.

When a test needs to look more deeply into the warnings, rather than just checking whether or not they occurred, code like this can be used:

```
with check_warnings(quiet=True) as w:
    warnings.warn("foo")
    assert str(w.args[0]) == "foo"
    warnings.warn("bar")
    assert str(w.args[0]) == "bar"
    assert str(w.warnings[0].args[0]) == "foo"
    assert str(w.warnings[1].args[0]) == "bar"
    w.reset()
    assert len(w.warnings) == 0
```

Aqui todos os avisos serão capturados e o código de teste testa os avisos diretamente capturados.

Alterado na versão 3.2: Novos argumentos opcionais *filters* e *quiet*.

`test.support.`**`check_no_resource_warning`**(*testcase*)
Context manager to check that no *ResourceWarning* was raised. You must remove the object which may emit *ResourceWarning* before the end of the context manager.

`test.support.`**`set_memlimit`**(*limit*)
Define os valores para `max_memuse' e :data:`real_max_memuse` para grandes testes de memória.

`test.support.`**`record_original_stdout`**(*stdout*)
Armazena o valor de *stdout*. Destina-se a manter o stdout no momento em que o registro começou.

`test.support.`**`get_original_stdout`**()
Retorna o stdout original definido por:func:*record_original_stdout* ou `sys.stdout` se não estiver definido.

`test.support.`**`strip_python_strerr`**(*stderr*)
Strip the *stderr* of a Python process from potential debug output emitted by the interpreter. This will typically be run on the result of *subprocess.Popen.communicate()*.

`test.support.`**`args_from_interpreter_flags`**()
Retorna uma lista de argumentos de linha de comando reproduzindo as configurações em `sys.flags` e `sys.warnoptions`.

`test.support.`**`optim_args_from_interpreter_flags`**()
Retorna a lista de argumentos da linha de comando reproduzindo as configurações de otimização atuais em `sys.flags`.

`test.support.`**`captured_stdin`**()
`test.support.`**`captured_stdout`**()
`test.support.`**`captured_stderr`**()
Um gerenciador de contexto que substitui temporariamente o fluxo nomeado pelo objeto *io.StringIO*.

Exemplo do uso com fluxos de saída:

```
with captured_stdout() as stdout, captured_stderr() as stderr:
    print("hello")
    print("error", file=sys.stderr)
assert stdout.getvalue() == "hello\n"
assert stderr.getvalue() == "error\n"
```

Exemplo de uso com fluxo de entrada:

```
with captured_stdin() as stdin:
    stdin.write('hello\n')
    stdin.seek(0)
    # call test code that consumes from sys.stdin
    captured = input()
self.assertEqual(captured, "hello")
```

test.support.**temp_dir**(*path=None*, *quiet=False*)
Um gerenciador de contexto que cria um diretório temporário no *path* e produz o diretório.

If *path* is `None`, the temporary directory is created using *tempfile.mkdtemp()*. If *quiet* is `False`, the context manager raises an exception on error. Otherwise, if *path* is specified and cannot be created, only a warning is issued.

test.support.**change_cwd**(*path*, *quiet=False*)
A context manager that temporarily changes the current working directory to *path* and yields the directory.

If *quiet* is `False`, the context manager raises an exception on error. Otherwise, it issues only a warning and keeps the current working directory the same.

test.support.**temp_cwd**(*name='tempcwd'*, *quiet=False*)
Um gerenciador de contexto que cria temporariamente um novo diretório e altera o diretório de trabalho atual (CWD).

The context manager creates a temporary directory in the current directory with name *name* before temporarily changing the current working directory. If *name* is `None`, the temporary directory is created using *tempfile.mkdtemp()*.

Se *quiet* é `False` e ele não possibilita criar ou alterar o CWD, um erro aparece. Por outro lado, somente um aviso surge e o CWD original é utilizado.

test.support.**temp_umask**(*umask*)
Um gerenciador de contexto que temporariamente define o processo de permissões de arquivos e diretórios , umask.

test.support.**transient_internet**(*resource_name*, ***, *timeout=30.0*, *errnos=()*)
A context manager that raises *ResourceDenied* when various issues with the internet connection manifest themselves as exceptions.

test.support.**disable_faulthandler**()
A context manager that replaces `sys.stderr` with `sys.__stderr__`.

test.support.**gc_collect**()
Force as many objects as possible to be collected. This is needed because timely deallocation is not guaranteed by the garbage collector. This means that `__del__` methods may be called later than expected and weakrefs may remain alive for longer than expected.

test.support.**disable_gc**()
A context manager that disables the garbage collector upon entry and reenables it upon exit.

test.support.**swap_attr**(*obj*, *attr*, *new_val*)
Context manager to swap out an attribute with a new object.

Utilização:

```
with swap_attr(obj, "attr", 5):
    ...
```

This will set `obj.attr` to 5 for the duration of the `with` block, restoring the old value at the end of the block. If `attr` doesn't exist on `obj`, it will be created and then deleted at the end of the block.

The old value (or `None` if it doesn't exist) will be assigned to the target of the "as" clause, if there is one.

`test.support.`**`swap_item`**(*obj*, *attr*, *new_val*)
Context manager to swap out an item with a new object.

Utilização:

```
with swap_item(obj, "item", 5):
    ...
```

This will set `obj["item"]` to 5 for the duration of the `with` block, restoring the old value at the end of the block. If `item` doesn't exist on `obj`, it will be created and then deleted at the end of the block.

The old value (or `None` if it doesn't exist) will be assigned to the target of the "as" clause, if there is one.

`test.support.`**`wait_threads_exit`**(*timeout=60.0*)
Context manager to wait until all threads created in the `with` statement exit.

`test.support.`**`start_threads`**(*threads*, *unlock=None*)
Context manager to start *threads*. It attempts to join the threads upon exit.

`test.support.`**`calcobjsize`**(*fmt*)
Return *`struct.calcsize()`* for `nP{fmt}0n` or, if `gettotalrefcount` exists, `2PnP{fmt}0P`.

`test.support.`**`calcvobjsize`**(*fmt*)
Return *`struct.calcsize()`* for `nPn{fmt}0n` or, if `gettotalrefcount` exists, `2PnPn{fmt}0P`.

`test.support.`**`checksizeof`**(*test*, *o*, *size*)
For testcase *test*, assert that the `sys.getsizeof` for *o* plus the GC header size equals *size*.

`test.support.`**`can_symlink`**()
Return `True` if the OS supports symbolic links, `False` otherwise.

`test.support.`**`can_xattr`**()
Return `True` if the OS supports xattr, `False` otherwise.

`@test.support.`**`skip_unless_symlink`**
Um decorador para executar testes que requerem suporte para links simbólicos.

`@test.support.`**`skip_unless_xattr`**
Um decorador para execução de testes que requerem suporte para xattr.

`@test.support.`**`skip_unless_bind_unix_socket`**
A decorator for running tests that require a functional bind() for Unix sockets.

`@test.support.`**`anticipate_failure`**(*condition*)
A decorator to conditionally mark tests with *`unittest.expectedFailure()`*. Any use of this decorator should have an associated comment identifying the relevant tracker issue.

`@test.support.`**`run_with_locale`**(*catstr*, **locales*)
A decorator for running a function in a different locale, correctly resetting it after it has finished. *catstr* is the locale category as a string (for example `"LC_ALL"`). The *locales* passed will be tried sequentially, and the first valid locale will be used.

@test.support.**run_with_tz**(*tz*)
A decorator for running a function in a specific timezone, correctly resetting it after it has finished.

@test.support.**requires_freebsd_version**(**min_version*)
Decorator for the minimum version when running test on FreeBSD. If the FreeBSD version is less than the minimum, raise *unittest.SkipTest*.

@test.support.**requires_linux_version**(**min_version*)
Decorator for the minimum version when running test on Linux. If the Linux version is less than the minimum, raise *unittest.SkipTest*.

@test.support.**requires_mac_version**(**min_version*)
Decorator for the minimum version when running test on Mac OS X. If the MAC OS X version is less than the minimum, raise *unittest.SkipTest*.

@test.support.**requires_IEEE_754**
Decorator for skipping tests on non-IEEE 754 platforms.

@test.support.**requires_zlib**
Decorator for skipping tests if *zlib* doesn't exist.

@test.support.**requires_gzip**
Decorator for skipping tests if *gzip* doesn't exist.

@test.support.**requires_bz2**
Decorator for skipping tests if *bz2* doesn't exist.

@test.support.**requires_lzma**
Decorator for skipping tests if *lzma* doesn't exist.

@test.support.**requires_resource**(*resource*)
Decorator for skipping tests if *resource* is not available.

@test.support.**requires_docstrings**
Decorator for only running the test if *HAVE_DOCSTRINGS*.

@test.support.**cpython_only**(*test*)
Decorator for tests only applicable to CPython.

@test.support.**impl_detail**(*msg=None*, ***guards*)
Decorator for invoking *check_impl_detail()* on *guards*. If that returns `False`, then uses *msg* as the reason for skipping the test.

@test.support.**no_tracing**(*func*)
Decorator to temporarily turn off tracing for the duration of the test.

@test.support.**refcount_test**(*test*)
Decorator for tests which involve reference counting. The decorator does not run the test if it is not run by CPython. Any trace function is unset for the duration of the test to prevent unexpected refcounts caused by the trace function.

@test.support.**reap_threads**(*func*)
Decorator to ensure the threads are cleaned up even if the test fails.

@test.support.**bigmemtest**(*size*, *memuse*, *dry_run=True*)
Decorator for bigmem tests.

size is a requested size for the test (in arbitrary, test-interpreted units.) *memuse* is the number of bytes per unit for the test, or a good estimate of it. For example, a test that needs two byte buffers, of 4 GiB each, could be decorated with `@bigmemtest(size=_4G, memuse=2)`.

The *size* argument is normally passed to the decorated test method as an extra argument. If *dry_run* is `True`, the value passed to the test method may be less than the requested value. If *dry_run* is `False`, it means the test doesn't support dummy runs when `-M` is not specified.

@test.support.**bigaddrspacetest**(*f*)
Decorator for tests that fill the address space. *f* is the function to wrap.

test.support.**make_bad_fd**()
Cria um descritor de arquivo inválido abrindo e fechando um arquivo temporário e retornando seu descritor.

test.support.**check_syntax_error**(*testcase*, *statement*, *errtext=''*, ***, *lineno=None*, *offset=None*)
Test for syntax errors in *statement* by attempting to compile *statement*. *testcase* is the *unittest* instance for the test. *errtext* is the text of the error raised by *SyntaxError*. If *lineno* is not None, compares to the line of the *SyntaxError*. If *offset* is not None, compares to the offset of the *SyntaxError*.

test.support.**open_urlresource**(*url*, **args*, ***kw*)
Abrir *url*. Se abrir falha, aumenta *TestFailed*.

test.support.**import_module**(*name*, *deprecated=False*, ***, *required_on()*)
This function imports and returns the named module. Unlike a normal import, this function raises *unittest.SkipTest* if the module cannot be imported.

Module and package deprecation messages are suppressed during this import if *deprecated* is `True`. If a module is required on a platform but optional for others, set *required_on* to an iterable of platform prefixes which will be compared against *sys.platform*.

Novo na versão 3.1.

test.support.**import_fresh_module**(*name*, *fresh=()*, *blocked=()*, *deprecated=False*)
This function imports and returns a fresh copy of the named Python module by removing the named module from `sys.modules` before doing the import. Note that unlike `reload()`, the original module is not affected by this operation.

fresh is an iterable of additional module names that are also removed from the `sys.modules` cache before doing the import.

blocked is an iterable of module names that are replaced with `None` in the module cache during the import to ensure that attempts to import them raise *ImportError*.

The named module and any modules named in the *fresh* and *blocked* parameters are saved before starting the import and then reinserted into `sys.modules` when the fresh import is complete.

Module and package deprecation messages are suppressed during this import if *deprecated* is `True`.

This function will raise *ImportError* if the named module cannot be imported.

Exemplo de uso:

```
# Get copies of the warnings module for testing without affecting the
# version being used by the rest of the test suite. One copy uses the
# C implementation, the other is forced to use the pure Python fallback
# implementation
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

Novo na versão 3.1.

test.support.**modules_setup**()
Retorna a cópia de *sys.modules*.

test.support.**modules_cleanup**(*oldmodules*)
Remove modules except for *oldmodules* and `encodings` in order to preserve internal cache.

test.support.**threading_setup**()
Return current thread count and copy of dangling threads.

test.support.**threading_cleanup**(**original_values*)
Cleanup up threads not specified in *original_values*. Designed to emit a warning if a test leaves running threads in the background.

test.support.**join_thread**(*thread*, *timeout=30.0*)
Join a *thread* within *timeout*. Raise an `AssertionError` if thread is still alive after *timeout* seconds.

test.support.**reap_children**()
Use this at the end of `test_main` whenever sub-processes are started. This will help ensure that no extra children (zombies) stick around to hog resources and create problems when looking for refleaks.

test.support.**get_attribute**(*obj*, *name*)
Get an attribute, raising `unittest.SkipTest` if `AttributeError` is raised.

test.support.**bind_port**(*sock*, *host=HOST*)
Bind the socket to a free port and return the port number. Relies on ephemeral ports in order to ensure we are using an unbound port. This is important as many tests may be running simultaneously, especially in a buildbot environment. This method raises an exception if the `sock.family` is `AF_INET` and `sock.type` is `SOCK_STREAM`, and the socket has `SO_REUSEADDR` or `SO_REUSEPORT` set on it. Tests should never set these socket options for TCP/IP sockets. The only case for setting these options is testing multicasting via multiple UDP sockets.

Additionally, if the `SO_EXCLUSIVEADDRUSE` socket option is available (i.e. on Windows), it will be set on the socket. This will prevent anyone else from binding to our host/port for the duration of the test.

test.support.**bind_unix_socket**(*sock*, *addr*)
Bind a unix socket, raising `unittest.SkipTest` if `PermissionError` is raised.

test.support.**find_unused_port**(*family=socket.AF_INET*, *socktype=socket.SOCK_STREAM*)
Returns an unused port that should be suitable for binding. This is achieved by creating a temporary socket with the same family and type as the `sock` parameter (default is `AF_INET`, `SOCK_STREAM`), and binding it to the specified host address (defaults to `0.0.0.0`) with the port set to 0, eliciting an unused ephemeral port from the OS. The temporary socket is then closed and deleted, and the ephemeral port is returned.

Either this method or `bind_port()` should be used for any tests where a server socket needs to be bound to a particular port for the duration of the test. Which one to use depends on whether the calling code is creating a Python socket, or if an unused port needs to be provided in a constructor or passed to an external program (i.e. the `-accept` argument to openssl's s_server mode). Always prefer `bind_port()` over `find_unused_port()` where possible. Using a hard coded port is discouraged since it can make multiple instances of the test impossible to run simultaneously, which is a problem for buildbots.

test.support.**load_package_tests**(*pkg_dir*, *loader*, *standard_tests*, *pattern*)
Generic implementation of the `unittest` `load_tests` protocol for use in test packages. *pkg_dir* is the root directory of the package; *loader*, *standard_tests*, and *pattern* are the arguments expected by `load_tests`. In simple cases, the test package's `__init__.py` can be the following:

```
import os
from test.support import load_package_tests

def load_tests(*args):
    return load_package_tests(os.path.dirname(__file__), *args)
```

test.support.**fs_is_case_insensitive**(*directory*)
Return `True` if the file system for *directory* is case-insensitive.

test.support.**detect_api_mismatch**(*ref_api*, *other_api*, ***, *ignore=()*)
Returns the set of attributes, functions or methods of *ref_api* not found on *other_api*, except for a defined list of items to be ignored in this check specified in *ignore*.

By default this skips private attributes beginning with '_' but includes all magic methods, i.e. those starting and ending in '__'.

Novo na versão 3.5.

test.support.**patch**(*test_instance*, *object_to_patch*, *attr_name*, *new_value*)
Override *object_to_patch.attr_name* with *new_value*. Also add cleanup procedure to *test_instance* to restore *object_to_patch* for *attr_name*. The *attr_name* should be a valid attribute for *object_to_patch*.

test.support.**run_in_subinterp**(*code*)
Run *code* in subinterpreter. Raise *unittest.SkipTest* if *tracemalloc* is enabled.

test.support.**check_free_after_iterating**(*test*, *iter*, *cls*, *args=()*)
Assert that *iter* is deallocated after iterating.

test.support.**missing_compiler_executable**(*cmd_names=[]*)
Check for the existence of the compiler executables whose names are listed in *cmd_names* or all the compiler executables when *cmd_names* is empty and return the first missing executable or `None` when none is found missing.

test.support.**check__all__**(*test_case*, *module*, *name_of_module=None*, *extra=()*, *blacklist=()*)
Assert that the `__all__` variable of *module* contains all public names.

The module's public names (its API) are detected automatically based on whether they match the public name convention and were defined in *module*.

The *name_of_module* argument can specify (as a string or tuple thereof) what module(s) an API could be defined in order to be detected as a public API. One case for this is when *module* imports part of its public API from other modules, possibly a C backend (like `csv` and its `_csv`).

The *extra* argument can be a set of names that wouldn't otherwise be automatically detected as "public", like objects without a proper `__module__` attribute. If provided, it will be added to the automatically detected ones.

The *blacklist* argument can be a set of names that must not be treated as part of the public API even though their names indicate otherwise.

Exemplo de uso:

```
import bar
import foo
import unittest
from test import support

class MiscTestCase(unittest.TestCase):
    def test__all__(self):
        support.check__all__(self, foo)

class OtherTestCase(unittest.TestCase):
    def test__all__(self):
        extra = {'BAR_CONST', 'FOO_CONST'}
        blacklist = {'baz'}  # Undocumented name.
        # bar imports part of its API from _bar.
        support.check__all__(self, bar, ('bar', '_bar'),
                             extra=extra, blacklist=blacklist)
```

Novo na versão 3.6.

test.support.**adjust_int_max_str_digits**(*max_digits*)
This function returns a context manager that will change the global *sys.set_int_max_str_digits()* setting for the duration of the context to allow execution of test code that needs a different limit on the number of digits when converting between an integer and string.

Novo na versão 3.7.14.

The *test.support* module defines the following classes:

class test.support.**TransientResource**(*exc*, ***kwargs*)
Instances are a context manager that raises *ResourceDenied* if the specified exception type is raised. Any keyword arguments are treated as attribute/value pairs to be compared against any exception raised within the `with` statement. Only if all pairs match properly against attributes on the exception is *ResourceDenied* raised.

class test.support.**EnvironmentVarGuard**
Class used to temporarily set or unset environment variables. Instances can be used as a context manager and have a complete dictionary interface for querying/modifying the underlying `os.environ`. After exit from the context manager all changes to environment variables done through this instance will be rolled back.

Alterado na versão 3.1: Adicionado interface de dicionário.

EnvironmentVarGuard.**set**(*envvar*, *value*)
Temporariamente define a variável de ambiente `envvar` para o valor `value`.

EnvironmentVarGuard.**unset**(*envvar*)
Desativa temporariamente a variável de ambiente `envvar`.

class test.support.**SuppressCrashReport**
A context manager used to try to prevent crash dialog popups on tests that are expected to crash a subprocess.

No Windows, desativa as caixas de diálogo Relatório de Erros do Windows usando SetErrorMode <https://msdn.microsoft.com/en-us/library/windows/desktop/ms680621.aspx>_.

On UNIX, *resource.setrlimit()* is used to set *resource.RLIMIT_CORE*'s soft limit to 0 to prevent coredump file creation.

On both platforms, the old value is restored by `__exit__()`.

class test.support.**CleanImport**(**module_names*)
A context manager to force import to return a new module reference. This is useful for testing module-level behaviors, such as the emission of a DeprecationWarning on import. Example usage:

```
with CleanImport('foo'):
    importlib.import_module('foo')  # New reference.
```

class test.support.**DirsOnSysPath**(**paths*)
A context manager to temporarily add directories to sys.path.

This makes a copy of *sys.path*, appends any directories given as positional arguments, then reverts *sys.path* to the copied settings when the context ends.

Note that *all* *sys.path* modifications in the body of the context manager, including replacement of the object, will be reverted at the end of the block.

class test.support.**SaveSignals**
Class to save and restore signal handlers registered by the Python signal handler.

class test.support.**Matcher**

matches(*self*, *d*, ***kwargs*)
Tente combinar um único dict com os argumentos fornecidos.

match_value(*self*, *k*, *dv*, *v*)
Tente combinar um único valor armazenado (*dv*) com um valor fornecido (*v*).

class test.support.**WarningsRecorder**
Class used to record warnings for unit tests. See documentation of *check_warnings()* above for more details.

class test.support.**BasicTestRunner**

run(*test*)
Executa *test* e retorna o resultado.

class test.support.**TestHandler**(*logging.handlers.BufferingHandler*)
Class for logging support.

class test.support.**FakePath**(*path*)
Simple *path-like object*. It implements the `__fspath__()` method which just returns the *path* argument. If *path* is an exception, it will be raised in `__fspath__()`.

27.10 test.support.script_helper — Utilities for the Python execution tests

The *test.support.script_helper* module provides support for Python's script execution tests.

test.support.script_helper.**interpreter_requires_environment**()
Return `True` if `sys.executable interpreter` requires environment variables in order to be able to run at all.

This is designed to be used with `@unittest.skipIf()` to annotate tests that need to use an `assert_python*()` function to launch an isolated mode (`-I`) or no environment mode (`-E`) sub-interpreter process.

A normal build & test does not run into this situation but it can happen when trying to run the standard library test suite from an interpreter that doesn't have an obvious home with Python's current home finding logic.

Setting `PYTHONHOME` is one way to get most of the testsuite to run in that situation. `PYTHONPATH` or `PYTHONUSERSITE` are other common environment variables that might impact whether or not the interpreter can start.

test.support.script_helper.**run_python_until_end**(**args*, ***env_vars*)
Set up the environment based on *env_vars* for running the interpreter in a subprocess. The values can include `__isolated`, `__cleanenv`, `__cwd`, and `TERM`.

test.support.script_helper.**assert_python_ok**(**args*, ***env_vars*)
Assert that running the interpreter with *args* and optional environment variables *env_vars* succeeds (`rc == 0`) and return a `(return code, stdout, stderr)` tuple.

If the `__cleanenv` keyword is set, *env_vars* is used as a fresh environment.

Python is started in isolated mode (command line option `-I`), except if the `__isolated` keyword is set to `False`.

test.support.script_helper.**assert_python_failure**(**args*, ***env_vars*)
Assert that running the interpreter with *args* and optional environment variables *env_vars* fails (`rc != 0`) and return a `(return code, stdout, stderr)` tuple.

See *assert_python_ok()* for more options.

test.support.script_helper.**spawn_python**(**args*, *stdout=subprocess.PIPE*, *stderr=subprocess.STDOUT*, ***kw*)
Run a Python subprocess with the given arguments.

kw is extra keyword args to pass to *subprocess.Popen()*. Returns a *subprocess.Popen* object.

test.support.script_helper.**kill_python**(*p*)
Run the given *subprocess.Popen* process until completion and return stdout.

test.support.script_helper.**make_script**(*script_dir*, *script_basename*, *source*, *omit_suffix=False*)
Cria um script contendo *source* no caminho *script_dir* e *script_basename*. Se *omit_suffix* `False`, acrescente `.py` ao nome. Retorna o caminho completo do script.

test.support.script_helper.**make_zip_script**(*zip_dir*, *zip_basename*, *script_name*, *name_in_zip=None*)
Cria um arquivo zip em *zip_dir* e *zip_basename* com a extensão `zip` que contém os arquivos em *script_name*. *name_in_zip* é o nome do arquivo. Retorna uma tupla contendo `(full path, full path of archive name)`.

test.support.script_helper.**make_pkg**(*pkg_dir*, *init_source=''*)
Cria um diretório nomeado *pkg_dir* contendo um arquivo `__init__` com *init_source* como seus conteúdos.

test.support.script_helper.**make_zip_pkg**(*zip_dir*, *zip_basename*, *pkg_name*, *script_basename*, *source*, *depth=1*, *compiled=False*)
Create a zip package directory with a path of *zip_dir* and *zip_basename* containing an empty `__init__` file and a file *script_basename* containing the *source*. If *compiled* is `True`, both source files will be compiled and added to the zip package. Return a tuple of the full zip path and the archive name for the zip file.

Veja também o modo de desenvolvimento do Python: a opção `-X dev` e a variável de ambiente `PYTHONDEVMODE`.

CAPÍTULO 28

Depuração e perfilamento

Essas bibliotecas ajudam você com o desenvolvimento em Python: o depurador permite percorrer o código, analisar valores em memória durante a execução e definir pontos de interrupção, etc., e o perfilador executa o código e fornece uma análise detalhada dos tempos de execução, permitindo que você identifique gargalos em seus programas.

28.1 bdb — Debugger framework

Código Fonte: Lib/bdb.py

The *bdb* module handles basic debugger functions, like setting breakpoints or managing execution via the debugger.

The following exception is defined:

exception bdb.**BdbQuit**
Exception raised by the *Bdb* class for quitting the debugger.

The *bdb* module also defines two classes:

class bdb.**Breakpoint**(*self*, *file*, *line*, *temporary=0*, *cond=None*, *funcname=None*)
This class implements temporary breakpoints, ignore counts, disabling and (re-)enabling, and conditionals.

Breakpoints are indexed by number through a list called `bpbynumber` and by `(file, line)` pairs through `bplist`. The former points to a single instance of class *Breakpoint*. The latter points to a list of such instances since there may be more than one breakpoint per line.

When creating a breakpoint, its associated filename should be in canonical form. If a *funcname* is defined, a breakpoint hit will be counted when the first line of that function is executed. A conditional breakpoint always counts a hit.

Breakpoint instances have the following methods:

deleteMe()
Delete the breakpoint from the list associated to a file/line. If it is the last breakpoint in that position, it also deletes the entry for the file/line.

enable()
Mark the breakpoint as enabled.

disable()
Mark the breakpoint as disabled.

bpformat()
Return a string with all the information about the breakpoint, nicely formatted:

- The breakpoint number.
- Se é um temporário ou não.
- Its file,line position.
- The condition that causes a break.
- If it must be ignored the next N times.
- The breakpoint hit count.

Novo na versão 3.2.

bpprint(*out=None*)
Print the output of *bpformat()* to the file *out*, or if it is `None`, to standard output.

class bdb.**Bdb**(*skip=None*)
The *Bdb* class acts as a generic Python debugger base class.

This class takes care of the details of the trace facility; a derived class should implement user interaction. The standard debugger class (*pdb.Pdb*) is an example.

The *skip* argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. Whether a frame is considered to originate in a certain module is determined by the `__name__` in the frame globals.

Novo na versão 3.1: O argumento *skip*.

The following methods of *Bdb* normally don't need to be overridden.

canonic(*filename*)
Auxiliary method for getting a filename in a canonical form, that is, as a case-normalized (on case-insensitive filesystems) absolute path, stripped of surrounding angle brackets.

reset()
Set the `botframe`, `stopframe`, `returnframe` and `quitting` attributes with values ready to start debugging.

trace_dispatch(*frame*, *event*, *arg*)
This function is installed as the trace function of debugged frames. Its return value is the new trace function (in most cases, that is, itself).

The default implementation decides how to dispatch a frame, depending on the type of event (passed as a string) that is about to be executed. *event* can be one of the following:

- `"line"`: A new line of code is going to be executed.
- `"call"`: A function is about to be called, or another code block entered.
- `"return"`: A function or other code block is about to return.
- `"exception"`: An exception has occurred.
- `"c_call"`: A C function is about to be called.
- `"c_return"`: A C function has returned.

- `"c_exception"`: A C function has raised an exception.

For the Python events, specialized functions (see below) are called. For the C events, no action is taken.

The *arg* parameter depends on the previous event.

See the documentation for *sys.settrace()* for more information on the trace function. For more information on code and frame objects, refer to types.

dispatch_line(*frame*)
If the debugger should stop on the current line, invoke the *user_line()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the `Bdb.quitting` flag is set (which can be set from *user_line()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_call(*frame*, *arg*)
If the debugger should stop on this function call, invoke the *user_call()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the `Bdb.quitting` flag is set (which can be set from *user_call()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_return(*frame*, *arg*)
If the debugger should stop on this function return, invoke the *user_return()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the `Bdb.quitting` flag is set (which can be set from *user_return()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_exception(*frame*, *arg*)
If the debugger should stop at this exception, invokes the *user_exception()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the `Bdb.quitting` flag is set (which can be set from *user_exception()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

Normally derived classes don't override the following methods, but they may if they want to redefine the definition of stopping and breakpoints.

stop_here(*frame*)
This method checks if the *frame* is somewhere below `botframe` in the call stack. `botframe` is the frame in which debugging started.

break_here(*frame*)
This method checks if there is a breakpoint in the filename and line belonging to *frame* or, at least, in the current function. If the breakpoint is a temporary one, this method deletes it.

break_anywhere(*frame*)
This method checks if there is a breakpoint in the filename of the current frame.

Derived classes should override these methods to gain control over debugger operation.

user_call(*frame*, *argument_list*)
This method is called from *dispatch_call()* when there is the possibility that a break might be necessary anywhere inside the called function.

user_line(*frame*)
This method is called from *dispatch_line()* when either *stop_here()* or *break_here()* yields `True`.

user_return(*frame*, *return_value*)
This method is called from *dispatch_return()* when *stop_here()* yields `True`.

user_exception(*frame*, *exc_info*)
This method is called from *dispatch_exception()* when *stop_here()* yields `True`.

do_clear(*arg*)
Handle how a breakpoint must be removed when it is a temporary one.

This method must be implemented by derived classes.

Derived classes and clients can call the following methods to affect the stepping state.

set_step()
Stop after one line of code.

set_next(*frame*)
Stop on the next line in or below the given frame.

set_return(*frame*)
Stop when returning from the given frame.

set_until(*frame*)
Stop when the line with the line no greater than the current one is reached or when returning from current frame.

set_trace([*frame*])
Start debugging from *frame*. If *frame* is not specified, debugging starts from caller's frame.

set_continue()
Stop only at breakpoints or when finished. If there are no breakpoints, set the system trace function to `None`.

set_quit()
Set the `quitting` attribute to `True`. This raises *BdbQuit* in the next call to one of the `dispatch_*()` methods.

Derived classes and clients can call the following methods to manipulate breakpoints. These methods return a string containing an error message if something went wrong, or `None` if all is well.

set_break(*filename*, *lineno*, *temporary=0*, *cond*, *funcname*)
Set a new breakpoint. If the *lineno* line doesn't exist for the *filename* passed as argument, return an error message. The *filename* should be in canonical form, as described in the *canonic()* method.

clear_break(*filename*, *lineno*)
Delete the breakpoints in *filename* and *lineno*. If none were set, an error message is returned.

clear_bpbynumber(*arg*)
Delete the breakpoint which has the index *arg* in the `Breakpoint.bpbynumber`. If *arg* is not numeric or out of range, return an error message.

clear_all_file_breaks(*filename*)
Delete all breakpoints in *filename*. If none were set, an error message is returned.

clear_all_breaks()
Delete all existing breakpoints.

get_bpbynumber(*arg*)
Return a breakpoint specified by the given number. If *arg* is a string, it will be converted to a number. If *arg* is a non-numeric string, if the given breakpoint never existed or has been deleted, a *ValueError* is raised.

Novo na versão 3.2.

get_break(*filename*, *lineno*)
Check if there is a breakpoint for *lineno* of *filename*.

get_breaks(*filename*, *lineno*)
Return all breakpoints for *lineno* in *filename*, or an empty list if none are set.

get_file_breaks(*filename*)
Return all breakpoints in *filename*, or an empty list if none are set.

get_all_breaks()
Return all breakpoints that are set.

Derived classes and clients can call the following methods to get a data structure representing a stack trace.

get_stack(*f*, *t*)
Get a list of records for a frame and all higher (calling) and lower frames, and the size of the higher part.

format_stack_entry(*frame_lineno*, *lprefix=': '*)
Return a string with information about a stack entry, identified by a `(frame, lineno)` tuple:

- The canonical form of the filename which contains the frame.
- The function name, or `"<lambda>"`.
- O argumento de entrada.
- The return value.
- The line of code (if it exists).

Os dois métodos a seguir podem ser chamados pelos clientes para usar um depurador e depurar um *statement*, fornecido como uma string.

run(*cmd*, *globals=None*, *locals=None*)
Debug a statement executed via the `exec()` function. *globals* defaults to `__main__.__dict__`, *locals* defaults to *globals*.

runeval(*expr*, *globals=None*, *locals=None*)
Debug an expression executed via the `eval()` function. *globals* and *locals* have the same meaning as in `run()`.

runctx(*cmd*, *globals*, *locals*)
For backwards compatibility. Calls the `run()` method.

runcall(*func*, **args*, ***kwds*)
Debug a single function call, and return its result.

Finally, the module defines the following functions:

bdb.**checkfuncname**(*b*, *frame*)
Check whether we should break here, depending on the way the breakpoint *b* was set.

If it was set via line number, it checks if `b.line` is the same as the one in the frame also passed as argument. If the breakpoint was set via function name, we have to check we are in the right frame (the right function) and if we are in its first executable line.

bdb.**effective**(*file*, *line*, *frame*)
Determine if there is an effective (active) breakpoint at this line of code. Return a tuple of the breakpoint and a boolean that indicates if it is ok to delete a temporary breakpoint. Return `(None, None)` if there is no matching breakpoint.

bdb.**set_trace**()
Start debugging with a `Bdb` instance from caller's frame.

28.2 faulthandler — Dump the Python traceback

Novo na versão 3.3.

This module contains functions to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal. Call *faulthandler.enable()* to install fault handlers for the SIGSEGV, SIGFPE, SIGABRT, SIGBUS, and SIGILL signals. You can also enable them at startup by setting the PYTHONFAULTHANDLER environment variable or by using the -X faulthandler command line option.

The fault handler is compatible with system fault handlers like Apport or the Windows fault handler. The module uses an alternative stack for signal handlers if the sigaltstack() function is available. This allows it to dump the traceback even on a stack overflow.

The fault handler is called on catastrophic cases and therefore can only use signal-safe functions (e.g. it cannot allocate memory on the heap). Because of this limitation traceback dumping is minimal compared to normal Python tracebacks:

- Only ASCII is supported. The backslashreplace error handler is used on encoding.
- Each string is limited to 500 characters.
- Only the filename, the function name and the line number are displayed. (no source code)
- It is limited to 100 frames and 100 threads.
- The order is reversed: the most recent call is shown first.

By default, the Python traceback is written to *sys.stderr*. To see tracebacks, applications must be run in the terminal. A log file can alternatively be passed to *faulthandler.enable()*.

The module is implemented in C, so tracebacks can be dumped on a crash or when Python is deadlocked.

28.2.1 Dumping the traceback

faulthandler.**dump_traceback**(*file=sys.stderr*, *all_threads=True*)
Dump the tracebacks of all threads into *file*. If *all_threads* is False, dump only the current thread.

Alterado na versão 3.5: Added support for passing file descriptor to this function.

28.2.2 Fault handler state

faulthandler.**enable**(*file=sys.stderr*, *all_threads=True*)
Enable the fault handler: install handlers for the SIGSEGV, SIGFPE, SIGABRT, SIGBUS and SIGILL signals to dump the Python traceback. If *all_threads* is True, produce tracebacks for every running thread. Otherwise, dump only the current thread.

The *file* must be kept open until the fault handler is disabled: see *issue with file descriptors*.

Alterado na versão 3.5: Added support for passing file descriptor to this function.

Alterado na versão 3.6: On Windows, a handler for Windows exception is also installed.

faulthandler.**disable**()
Disable the fault handler: uninstall the signal handlers installed by *enable()*.

faulthandler.**is_enabled**()
Check if the fault handler is enabled.

28.2.3 Dumping the tracebacks after a timeout

faulthandler.**dump_traceback_later**(*timeout*, *repeat=False*, *file=sys.stderr*, *exit=False*)
Dump the tracebacks of all threads, after a timeout of *timeout* seconds, or every *timeout* seconds if *repeat* is `True`. If *exit* is `True`, call `_exit()` with status=1 after dumping the tracebacks. (Note `_exit()` exits the process immediately, which means it doesn't do any cleanup like flushing file buffers.) If the function is called twice, the new call replaces previous parameters and resets the timeout. The timer has a sub-second resolution.

The *file* must be kept open until the traceback is dumped or *cancel_dump_traceback_later()* is called: see *issue with file descriptors*.

This function is implemented using a watchdog thread.

Alterado na versão 3.7: This function is now always available.

Alterado na versão 3.5: Added support for passing file descriptor to this function.

faulthandler.**cancel_dump_traceback_later**()
Cancel the last call to *dump_traceback_later()*.

28.2.4 Dumping the traceback on a user signal

faulthandler.**register**(*signum*, *file=sys.stderr*, *all_threads=True*, *chain=False*)
Register a user signal: install a handler for the *signum* signal to dump the traceback of all threads, or of the current thread if *all_threads* is `False`, into *file*. Call the previous handler if chain is `True`.

The *file* must be kept open until the signal is unregistered by *unregister()*: see *issue with file descriptors*.

Não disponível no Windows.

Alterado na versão 3.5: Added support for passing file descriptor to this function.

faulthandler.**unregister**(*signum*)
Unregister a user signal: uninstall the handler of the *signum* signal installed by *register()*. Return `True` if the signal was registered, `False` otherwise.

Não disponível no Windows.

28.2.5 Issue with file descriptors

enable(), *dump_traceback_later()* and *register()* keep the file descriptor of their *file* argument. If the file is closed and its file descriptor is reused by a new file, or if *os.dup2()* is used to replace the file descriptor, the traceback will be written into a different file. Call these functions again each time that the file is replaced.

28.2.6 Exemplo

Example of a segmentation fault on Linux with and without enabling the fault handler:

```
$ python3 -c "import ctypes; ctypes.string_at(0)"
Segmentation fault

$ python3 -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault
```

(continua na próxima página)

(continuação da página anterior)

```
Current thread 0x00007fb899f39700 (most recent call first):
  File "/home/python/cpython/Lib/ctypes/__init__.py", line 486 in string_at
  File "<stdin>", line 1 in <module>
Segmentation fault
```

28.3 pdb — O Depurador do Python

Código Fonte: Lib/pdb.py

O módulo *pdb* define um depurador de código-fonte interativo para programas Python. Ele possui suporte a definição de pontos de interrupção (condicionais) e passo único no nível da linha de origem, inspeção de quadros de pilha, listagem de código-fonte e avaliação de código Python arbitrário no contexto de qualquer quadro de pilha. Ele também tem suporte a depuração *post-mortem* e pode ser chamado sob controle do programa.

O depurador é extensível – na verdade, ele é definido como a classe *Pdb*. Atualmente, isso não está documentado, mas é facilmente compreendido pela leitura do código-fonte. A interface de extensão usa os módulos *bdb* e *cmd*.

O prompt do depurador é `(Pdb)`. O uso típico para executar um programa sob controle do depurador é:

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

Alterado na versão 3.3: O preenchimento por tabulação através do módulo *readline* está disponível para comandos e argumentos de comando, por exemplo os nomes globais e locais atuais são oferecidos como argumentos do comando `p`.

`pdb.py` também pode ser chamado como um script para depurar outros scripts. Por exemplo:

```
python3 -m pdb myscript.py
```

Quando invocado como um script, o pdb entra automaticamente na depuração *post-mortem* se o programa que está sendo depurado for encerrado de forma anormal. Após a depuração *post-mortem* (ou após a saída normal do programa), o pdb reiniciará o programa. A reinicialização automática preserva o estado do pdb (p.ex., pontos de interrupção) e, na maioria dos casos, é mais útil do que encerrar o depurador na saída do programa.

Novo na versão 3.2: `pdb.py` agora aceita uma opção `-c` que executa comandos como se fossem dados em um arquivo `.pdbrc`. Veja *Comandos de depuração*.

Novo na versão 3.7: `pdb.py` agora aceita uma opção `-m` que executa módulos de maneira semelhante à de `python3 -m`. Como em um script, o depurador fará uma pausa na execução imediatamente antes da primeira linha do módulo.

O uso típico para invadir o depurador a partir de um programa em execução é inserir

```
import pdb; pdb.set_trace()
```

no local em que você deseja interromper o depurador. Em seguida, você pode percorrer o código após esta instrução e continuar executando sem o depurador usando o comando *continue*.

Novo na versão 3.7: A função embutida *breakpoint()*, quando chamada com valores padrão, pode ser usada em vez de `import pdb; pdb.set_trace()`.

O uso típico para inspecionar um programa com falha é:

```
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test2
    print(spam)
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print(spam)
(Pdb)
```

O módulo define as seguintes funções; cada uma entra no depurador de uma maneira ligeiramente diferente:

pdb.**run**(*statement*, *globals=None*, *locals=None*)
: Executa a instrução *statement* (fornecida como uma string ou um objeto código) sob controle do depurador. O prompt do depurador aparece antes de qualquer código ser executado; você pode definir pontos de interrupção e digitar *continue* ou pode percorrer a instrução usando *step* ou *next* (todos esses comandos são explicados abaixo). Os argumentos opcionais *globals* e *locals* especificam o ambiente em que o código é executado; por padrão, o dicionário do módulo *__main__* é usado. (Veja a explicação das funções embutidas *exec()* ou *eval()*.)

pdb.**runeval**(*expression*, *globals=None*, *locals=None*)
: Avalia a expressão *expression* (fornecida como uma string ou um objeto código) sob controle do depurador. Quando *runeval()* retorna, ele retorna o valor da expressão. Caso contrário, esta função é semelhante a *run()*.

pdb.**runcall**(*function*, **args*, ***kwds*)
: Chama a função *function* (um objeto função ou método, não uma string) com os argumentos fornecidos. Quando *runcall()* retorna, ele retorna qualquer que seja a chamada de função retornada. O prompt do depurador aparece assim que a função é inserida.

pdb.**set_trace**(*, *header=None*)
: Entra no depurador no quadro da pilha de chamada. Isso é útil para codificar um ponto de interrupção em um determinado ponto de um programa, mesmo que o código não esteja sendo depurado de outra forma (por exemplo, quando uma asserção falha). Se fornecido, *header* é impresso no console imediatamente antes do início da depuração.

  Alterado na versão 3.7: O argumento somente-nomeado *header*.

pdb.**post_mortem**(*traceback=None*)
: Entra na depuração *post-mortem* do objeto *traceback* fornecido. Se não for fornecido um *traceback*, será usada a exceção que está sendo manipulada no momento (uma exceção deve ser manipulada para que o padrão seja usado).

pdb.**pm**()
: Entra na depuração *post-mortem* do traceback encontrado em *sys.last_traceback*.

As funções `run*` e a *set_trace()* são aliases, ou apelidos, para instanciar a classe *Pdb* e chamar o método com o mesmo nome. Se você deseja acessar outros recursos, faça você mesmo:

class pdb.**Pdb**(*completekey='tab'*, *stdin=None*, *stdout=None*, *skip=None*, *nosigint=False*, *readrc=True*)
: *Pdb* é a classe do depurador.

  Os argumentos *completekey*, *stdin* e *stdout* são passados para a classe subjacente *cmd.Cmd*; veja a descrição lá.

O argumento *skip*, se fornecido, deve ser um iterável de padrões de nome de módulo no estilo glob. O depurador não entrará nos quadros que se originam em um módulo que corresponde a um desses padrões.[1]

Por padrão, o Pdb define um manipulador para o sinal SIGINT (que é enviado quando o usuário pressiona `Ctrl-C` no console) quando você dá um comando `continue`. Isso permite que você entre no depurador novamente pressionando `Ctrl-C`. Se você deseja que o Pdb não toque no manipulador SIGINT, defina *nosigint* como true.

O argumento *readrc* é padronizado como true e controla se o Pdb carregará arquivos .pdbrc do sistema de arquivos.

Exemplo de chamada para habilitar rastreamento com *skip*:

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

Novo na versão 3.1: O argumento *skip*.

Novo na versão 3.2: O argumento *nosigint*. Anteriormente, um manipulador de SIGINT nunca era definido por Pdb.

Alterado na versão 3.6: O argumento *readrc*.

run(*statement*, *globals=None*, *locals=None*)
runeval(*expression*, *globals=None*, *locals=None*)
runcall(*function*, **args*, ***kwds*)
set_trace()
Consulte a documentação para as funções explicadas acima.

28.3.1 Comandos de depuração

Os comandos reconhecidos pelo depurador estão listados abaixo. A maioria dos comandos pode ser abreviada para uma ou duas letras, conforme indicado; por exemplo, `h(elp)` significa que `h` ou `help` podem ser usados para inserir o comando de ajuda (mas não `he` ou `hel`, nem `H` ou `Help` ou `HELP`). Os argumentos para os comandos devem ser separados por espaços em branco (espaços ou tabulações). Os argumentos opcionais estão entre colchetes (`[]`) na sintaxe do comando; os colchetes não devem ser digitados. As alternativas na sintaxe de comando são separadas por uma barra vertical (`|`).

Digitar uma linha em branco repete o último comando digitado. Exceção: se o último comando foi um comando *list*, as próximas 11 linhas serão listadas.

Os comandos que o depurador não reconhece são presumidos como instruções Python e são executados no contexto do programa que está sendo depurado. As instruções Python também podem ser prefixadas com um ponto de exclamação (`!`). Essa é uma maneira poderosa de inspecionar o programa que está sendo depurado; é até possível alterar uma variável ou chamar uma função. Quando ocorre uma exceção em uma instrução, o nome da exceção é impresso, mas o estado do depurador não é alterado.

O depurador possui suporte a *aliases*. Os aliases podem ter parâmetros que permitem um certo nível de adaptabilidade ao contexto em exame.

Vários comandos podem ser inseridos em uma única linha, separados por `;;`. (Um único `;` não é usado, pois é o separador de vários comandos em uma linha que é passada para o analisador sintático do Python.) Nenhuma inteligência é aplicada para separar os comandos; a entrada é dividida no primeiro par `;;`, mesmo que esteja no meio de uma string entre aspas.

Se um arquivo `.pdbrc` existir no diretório inicial do usuário ou no diretório atual, ele será lido e executado como se tivesse sido digitado no prompt do depurador. Isso é particularmente útil para aliases. Se os dois arquivos existirem, aquele no diretório inicial será lido primeiro e os aliases definidos poderão ser substituídos pelo arquivo local.

Alterado na versão 3.2: `.pdbrc` agora pode conter comandos que continuam a depuração, como *continue* ou *next*. Anteriormente, esses comandos não tinham efeito.

[1] Se um quadro é considerado originário de um determinado módulo é determinado pelo `__name__` nos globais do quadro.

h(elp) [command]
Sem argumento, imprime a lista de comandos disponíveis. Com um *command* como argumento, imprime ajuda sobre esse comando. `help pdb` exibe a documentação completa (a docstring do módulo *pdb*). Como o argumento *command* deve ser um identificador, `help exec` deve ser inserido para obter ajuda sobre o comando `!`.

w(here)
Exibe um stack trace (situação da pilha de execução), com o quadro mais recente na parte inferior. Uma seta indica o quadro atual, que determina o contexto da maioria dos comandos.

d(own) [count]
Move os níveis do quadro atual *count* (padrão 1) para baixo no stack trace (para um quadro mais recente).

u(p) [count]
Move os níveis do quadro atual na *count* (padrão 1) para cima no stack trace (para um quadro mais antigo).

b(reak) [([filename:]lineno | function) [, condition]]
Com um argumento *lineno*, define uma interrupção no arquivo atual. Com um argumento *function*, defina uma quebra na primeira instrução executável dentro dessa função. O número da linha pode ser prefixado com um nome de arquivo e dois pontos, para especificar um ponto de interrupção em outro arquivo (provavelmente um que ainda não foi carregado). O arquivo é pesquisado em *sys.path*. Observe que cada ponto de interrupção recebe um número ao qual todos os outros comandos de ponto de interrupção se referem.

Se um segundo argumento estiver presente, é uma expressão que deve ser avaliada como verdadeira antes que o ponto de interrupção seja respeitado.

Sem argumento, lista todas as quebras, inclusive para cada ponto de interrupção, o número de vezes que o ponto de interrupção foi atingido, a contagem atual de ignorados e a condição associada, se houver.

tbreak [([filename:]lineno | function) [, condition]]
Ponto de interrupção temporário, que é removido automaticamente quando é atingido pela primeira vez. Os argumentos são os mesmos que para *break*.

cl(ear) [filename:lineno | bpnumber [bpnumber ...]]
Com um argumento *filename:lineno*, limpa todos os pontos de interrupção nessa linha. Com uma lista separada por espaços de números de ponto de interrupção, limpa esses pontos de interrupção. Sem argumento, limpa todas as quebras (mas primeiro pede a confirmação).

disable [bpnumber [bpnumber ...]]
Desativa os pontos de interrupção fornecidos como uma lista separada por espaços de números de pontos de interrupção. Desabilitar um ponto de interrupção significa que ele não pode interromper a execução do programa, mas, ao contrário de limpar um ponto de interrupção, ele permanece na lista de pontos de interrupção e pode ser (re)ativado.

enable [bpnumber [bpnumber ...]]
Ativa o ponto de interrupção especificado.

ignore bpnumber [count]
Define a contagem de ignorados para o número do ponto de interrupção especificado. Se a contagem for omitida, a contagem de ignorados será definida como 0. Um ponto de interrupção se torna ativo quando a contagem de ignorados é zero. Quando diferente de zero, a contagem é decrementada cada vez que o ponto de interrupção é atingido e o ponto de interrupção não é desativado e qualquer condição associada é avaliada como verdadeira.

condition bpnumber [condition]
Define uma nova *condition* para o ponto de interrupção, uma expressão que deve ser avaliada como verdadeira antes que o ponto de interrupção seja respeitado. Se *condition* for omitida, qualquer condição existente será removida; isto é, o ponto de interrupção é tornado incondicional.

commands [bpnumber]

Especifica uma lista de comandos para o número do ponto de interrupção *bpnumber*. Os próprios comandos aparecem nas seguintes linhas. Digite em uma linha contendo apenas `end` para finalizar os comandos. Um exemplo:

```
(Pdb) commands 1
(com) p some_variable
(com) end
(Pdb)
```

Para remover todos os comandos de um ponto de interrupção, digite `commands` e siga-o imediatamente com `end`; isto é, não dê comandos.

Sem argumento *bpnumber*, `commands` refere-se ao último conjunto de pontos de interrupção.

Você pode usar comandos de ponto de interrupção para iniciar seu programa novamente. Simplesmente use o comando *continue*, ou *step*, ou qualquer outro comando que reinicie a execução.

Especificar qualquer comando que retome a execução (atualmente *continue*, *step*, *next*, *return*, *jump*, *quit* e suas abreviações) finaliza a lista de comandos (como se esse comando fosse imediatamente seguido pelo final). Isso ocorre sempre que você retoma a execução (mesmo com uma simples etapa ou etapa), você pode encontrar outro ponto de interrupção — que pode ter sua própria lista de comandos, levando a ambiguidades sobre qual lista executar.

Se você usar o comando 'silent' na lista de comandos, a mensagem usual sobre a parada em um ponto de interrupção não será impressa. Isso pode ser desejável para pontos de interrupção que devem imprimir uma mensagem específica e continuar. Se nenhum dos outros comandos imprimir alguma coisa, você não vê sinal de que o ponto de interrupção foi atingido.

s(tep)
Executa a linha atual, interrompe na primeira ocasião possível (em uma função chamada ou na próxima linha na função atual).

n(ext)
Continua a execução até que a próxima linha na função atual seja atingida ou ela retorne. (A diferença entre *next* e *step* é que *step* para dentro de uma função chamada, enquanto *next* executa funções chamadas em (quase) velocidade máxima, parando apenas na próxima linha da função atual.)

unt(il) `[lineno]`
Sem argumento, continua a execução até que a linha com um número maior que o atual seja atingida.

Com um número de linha, continua a execução até que uma linha com um número maior ou igual ao alcançado. Nos dois casos, também interrompe quando o quadro atual retornar.

Alterado na versão 3.2: Permite fornecer um número de linha explícito.

r(eturn)
Continua a execução até que a função atual retorne.

c(ont(inue))
Continua a execução, interrompe apenas quando um ponto de interrupção for encontrado.

j(ump) `lineno`
Define a próxima linha que será executada. Disponível apenas no quadro mais inferior. Isso permite voltar e executar o código novamente ou avançar para pular o código que você não deseja executar.

Deve-se notar que nem todos os saltos são permitidos – por exemplo, não é possível pular para o meio de um loop de `for` ou sair de uma cláusula `finally`.

l(ist) `[first[, last]]`
Lista o código-fonte do arquivo atual. Sem argumentos, lista 11 linhas ao redor da linha atual ou continue a listagem anterior. Com `.` como argumento, lista 11 linhas ao redor da linha atual. Com um argumento, lista 11 linhas nessa

linha. Com dois argumentos, lista o intervalo especificado; se o segundo argumento for menor que o primeiro, ele será interpretado como uma contagem.

A linha atual no quadro atual é indicada por `->`. Se uma exceção estiver sendo depurada, a linha em que a exceção foi originalmente gerada ou propagada é indicada por `>>`, se for diferente da linha atual.

Novo na versão 3.2: O marcador `>>`.

ll | longlist
Lista todo o código-fonte da função ou quadro atual. As linhas interessantes estão marcadas como para *list*.

Novo na versão 3.2.

a(rgs)
Imprime a lista de argumentos da função atual.

p expression
Avalia a expressão *expression* no contexto atual e imprima seu valor.

Nota: `print()` também pode ser usado, mas não é um comando de depuração — isso executa a função Python *print()*.

---

pp expression
Como o comando *p*, exceto que o valor da expressão é bastante impresso usando o módulo *pprint*.

whatis expression
Exibe o tipo da expressão *expression*.

source expression
Tenta obter o código-fonte para o objeto especificado e exibe-o.

Novo na versão 3.2.

display [expression]
Exibe o valor da expressão caso ela tenha sido alterada, sempre que a execução for interrompida no quadro atual.

Sem expressão, lista todas as expressões de exibição para o quadro atual.

Novo na versão 3.2.

undisplay [expression]
Não exibe mais a expressão no quadro atual. Sem expressão, limpa todas as expressões de exibição para o quadro atual.

Novo na versão 3.2.

interact
Inicia um interpretador interativo (usando o módulo *code*) cujo espaço de nomes global contenha todos os nomes (globais e locais) encontrados no escopo atual.

Novo na versão 3.2.

alias [name [command]]
Cria um alias chamado *name* que executa *command*. O comando *não* deve estar entre aspas. Os parâmetros substituíveis podem ser indicados por `%1`, `%2` e assim por diante, enquanto `%*` é substituído por todos os parâmetros. Se nenhum comando for fornecido, o alias atual para *name* será mostrado. Se nenhum argumento for fornecido, todos os aliases serão listados.

Os aliases podem ser aninhados e podem conter qualquer coisa que possa ser digitada legalmente no prompt do pdb. Observe que os comandos internos do pdb *podem* ser substituídos por aliases. Esse comando é oculto até que o alias seja removido. O alias é aplicado recursivamente à primeira palavra da linha de comando; todas as outras palavras da linha são deixadas em paz.

Como exemplo, aqui estão dois aliases úteis (especialmente quando colocados no arquivo `.pdbrc`):

```
# Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print("%1.",k,"=",%1.__dict__[k])
# Print instance variables in self
alias ps pi self
```

unalias name
Executa o alias especificado.

! statement
Executa a instrução *statement* (de uma só linha) no contexto do quadro de pilha atual. O ponto de exclamação pode ser omitido, a menos que a primeira palavra da instrução seja semelhante a um comando de depuração. Para definir uma variável global, você pode prefixar o comando de atribuição com uma instrução `global` na mesma linha, por exemplo:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

run [args ...]
restart [args ...]
Reinicia o programa Python depurado. Se um argumento é fornecido, ele é dividido com *shlex* e o resultado é usado como o novo *sys.argv*. Histórico, pontos de interrupção, ações e opções do depurador são preservados. *restart* é um apelido para *run*.

q(uit)
Sai do depurador. O programa que está sendo executado é abortado.

debug code
Entra em um depurador recursivo que percorre o argumento do código (que é uma expressão ou instrução arbitrária a ser executada no ambiente atual).

retval
Print the return value for the last return of a function.

28.4 The Python Profilers

Código Fonte: Lib/profile.py and Lib/pstats.py

28.4.1 Introduction to the profilers

cProfile and *profile* provide *deterministic profiling* of Python programs. A *profile* is a set of statistics that describes how often and for how long various parts of the program executed. These statistics can be formatted into reports via the *pstats* module.

The Python standard library provides two different implementations of the same profiling interface:

1. *cProfile* is recommended for most users; it's a C extension with reasonable overhead that makes it suitable for profiling long-running programs. Based on `lsprof`, contributed by Brett Rosen and Ted Czotter.
2. *profile*, a pure Python module whose interface is imitated by *cProfile*, but which adds significant overhead to profiled programs. If you're trying to extend the profiler in some way, the task might be easier with this module. Originally designed and written by Jim Roskind.

Nota: The profiler modules are designed to provide an execution profile for a given program, not for benchmarking purposes (for that, there is *timeit* for reasonably accurate results). This particularly applies to benchmarking Python code against C code: the profilers introduce overhead for Python code, but not for C-level functions, and so the C code would seem faster than any Python one.

28.4.2 Instant User's Manual

This section is provided for users that "don't want to read the manual." It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application.

To profile a function that takes a single argument, you can do:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

(Use *profile* instead of *cProfile* if the latter is not available on your system.)

The above action would run *re.compile()* and print profile results like the following:

```
      197 function calls (192 primitive calls) in 0.002 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 re.py:212(compile)
     1    0.000    0.000    0.001    0.001 re.py:268(_compile)
     1    0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
     1    0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
     4    0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
   3/1    0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

The first line indicates that 197 calls were monitored. Of those calls, 192 were *primitive*, meaning that the call was not induced via recursion. The next line: `Ordered by: standard name`, indicates that the text string in the far right column was used to sort the output. The column headings include:

ncalls for the number of calls.

tottime for the total time spent in the given function (and excluding time made in calls to sub-functions)

percall is the quotient of `tottime` divided by `ncalls`

cumtime is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate *even* for recursive functions.

percall is the quotient of `cumtime` divided by primitive calls

filename:lineno(function) provides the respective data of each function

When there are two numbers in the first column (for example `3/1`), it means that the function recursed. The second value is the number of primitive calls and the former is the total number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

Instead of printing the output at the end of the profile run, you can save the results to a file by specifying a filename to the `run()` function:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

The *pstats.Stats* class reads profile results from a file and formats them in various ways.

The file *cProfile* can also be invoked as a script to profile another script. For example:

```
python -m cProfile [-o output_file] [-s sort_order] (-m module | myscript.py)
```

`-o` writes the profile results to a file instead of to stdout

`-s` specifies one of the *sort_stats()* sort values to sort the output by. This only applies when `-o` is not supplied.

`-m` specifies that a module is being profiled instead of a script.

Novo na versão 3.7: Added the `-m` option.

The *pstats* module's *Stats* class has a variety of methods for manipulating and printing the data saved into a profile results file:

```
import pstats
from pstats import SortKey
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

The *strip_dirs()* method removed the extraneous path from all the module names. The *sort_stats()* method sorted all the entries according to the standard module/line/name string that is printed. The *print_stats()* method printed out all the statistics. You might try the following sort calls:

```
p.sort_stats(SortKey.NAME)
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```
p.sort_stats(SortKey.CUMULATIVE).print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```
p.sort_stats(SortKey.TIME).print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

```
p.sort_stats(SortKey.FILENAME).print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with `__init__` in them). As one final example, you could try:

```
p.sort_stats(SortKey.TIME, SortKey.CUMULATIVE).print_stats(.5, 'init')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: `.5`) of its original size, then only lines containing `init` are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (`p` is still sorted according to the last criteria) do:

```
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you're going to have to read the manual, or guess what the following functions do:

```
p.print_callees()
p.add('restats')
```

Invoked as a script, the *pstats* module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using *cmd*) and interactive help.

28.4.3 `profile` and `cProfile` Module Reference

Both the *profile* and *cProfile* modules provide the following functions:

`profile.`**`run`**(*command*, *filename=None*, *sort=-1*)

This function takes a single argument that can be passed to the *exec()* function, and an optional file name. In all cases this routine executes:

```
exec(command, __main__.__dict__, __main__.__dict__)
```

and gathers profiling statistics from the execution. If no file name is present, then this function automatically creates a *Stats* instance and prints a simple profiling report. If the sort value is specified, it is passed to this *Stats* instance to control how the results are sorted.

`profile.`**`runctx`**(*command*, *globals*, *locals*, *filename=None*, *sort=-1*)

This function is similar to *run()*, with added arguments to supply the globals and locals dictionaries for the *command* string. This routine executes:

```
exec(command, globals, locals)
```

and gathers profiling statistics as in the *run()* function above.

`class` `profile.`**`Profile`**(*timer=None*, *timeunit=0.0*, *subcalls=True*, *builtins=True*)

This class is normally only used if more precise control over profiling is needed than what the `cProfile.run()` function provides.

A custom timer can be supplied for measuring how long code takes to run via the *timer* argument. This must be a function that returns a single number representing the current time. If the number is an integer, the *timeunit* specifies a multiplier that specifies the duration of each unit of time. For example, if the timer returns times measured in thousands of seconds, the time unit would be `.001`.

Directly using the *Profile* class allows formatting profile results without writing the profile data to a file:

```
import cProfile, pstats, io
from pstats import SortKey
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
s = io.StringIO()
sortby = SortKey.CUMULATIVE
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
```

(continua na próxima página)

(continuação da página anterior)

```
ps.print_stats()
print(s.getvalue())
```

enable()
Start collecting profiling data.

disable()
Stop collecting profiling data.

create_stats()
Stop collecting profiling data and record the results internally as the current profile.

print_stats(*sort=-1*)
Cria um objeto *Stats* com base no perfil atual e imprime os resultados para stdout.

dump_stats(*filename*)
Write the results of the current profile to *filename*.

run(*cmd*)
Profile the cmd via *exec()*.

runctx(*cmd*, *globals*, *locals*)
Profile the cmd via *exec()* with the specified global and local environment.

runcall(*func*, **args*, ***kwargs*)
Profile `func(*args, **kwargs)`

Note that profiling will only work if the called command/function actually returns. If the interpreter is terminated (e.g. via a *sys.exit()* call during the called command/function execution) no profiling results will be printed.

28.4.4 The `Stats` Class

Analysis of the profiler data is done using the *Stats* class.

class pstats.**Stats**(**filenames or profile*, *stream=sys.stdout*)
This class constructor creates an instance of a "statistics object" from a *filename* (or list of filenames) or from a `Profile` instance. Output will be printed to the stream specified by *stream*.

The file selected by the above constructor must have been created by the corresponding version of *profile* or *cProfile*. To be specific, there is *no* file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers, or the same profiler run on a different operating system. If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing *Stats* object, the *add()* method can be used.

Instead of reading the profile data from a file, a `cProfile.Profile` or *profile.Profile* object can be used as the profile data source.

Stats objects have the following methods:

strip_dirs()
This method for the *Stats* class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a "random" order, as it was just after object initialization and loading. If *strip_dirs()* causes two function names to be indistinguishable (they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

add(**filenames*)
This method of the *Stats* class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of *profile.run()* or `cProfile.run()`. Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

dump_stats(*filename*)
Save the data loaded into the *Stats* object to a file named *filename*. The file is created if it does not exist, and is overwritten if it already exists. This is equivalent to the method of the same name on the *profile.Profile* and `cProfile.Profile` classes.

sort_stats(**keys*)
This method modifies the *Stats* object by sorting it according to the supplied criteria. The argument can be either a string or a SortKey enum identifying the basis of a sort (example: `'time'`, `'name'`, `SortKey.TIME` or `SortKey.NAME`). The SortKey enums argument have advantage over the string argument in that it is more robust and less error prone.

When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, `sort_stats(SortKey.NAME, SortKey.FILE)` will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

For the string argument, abbreviations can be used for any key names, as long as the abbreviation is unambiguous.

The following are the valid string and SortKey:

Valid String Arg	Valid enum Arg	Significado
`'calls'`	SortKey.CALLS	call count
`'cumulative'`	SortKey.CUMULATIVE	cumulative time
`'cumtime'`	N/D	cumulative time
`'file'`	N/D	file name
`'filename'`	SortKey.FILENAME	file name
`'module'`	N/D	file name
`'ncalls'`	N/D	call count
`'pcalls'`	SortKey.PCALLS	primitive call count
`'line'`	SortKey.LINE	line number
`'name'`	SortKey.NAME	function name
`'nfl'`	SortKey.NFL	name/file/line
`'stdname'`	SortKey.STDNAME	standard name
`'time'`	SortKey.TIME	internal time
`'tottime'`	N/D	internal time

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (alphabetical). The subtle distinction between `SortKey.NFL` and `SortKey.STDNAME` is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, `SortKey.NFL` does a numeric compare of the line numbers. In fact, `sort_stats(SortKey.NFL)` is the same as `sort_stats(SortKey.NAME, SortKey.FILENAME, SortKey.LINE)`.

For backward-compatibility reasons, the numeric arguments `-1`, `0`, `1`, and `2` are permitted. They are interpreted as `'stdname'`, `'calls'`, `'time'`, and `'cumulative'` respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

Novo na versão 3.7: Added the SortKey enum.

reverse_order()
This method for the *Stats* class reverses the ordering of the basic list within the object. Note that by default ascending vs descending order is properly selected based on the sort key of choice.

print_stats(**restrictions*)
This method for the *Stats* class prints out a report as described in the *profile.run()* definition.

The order of the printing is based on the last *sort_stats()* operation done on the object (subject to caveats in *add()* and *strip_dirs()*).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a string that will interpreted as a regular expression (to pattern match the standard name that is printed). If several restrictions are provided, then they are applied sequentially. For example:

```
print_stats(.1, 'foo:')
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename `.*foo:`. In contrast, the command:

```
print_stats('foo:', .1)
```

would limit the list to all functions having file names `.*foo:`, and then proceed to only print the first 10% of them.

print_callers(**restrictions*)
This method for the *Stats* class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by *print_stats()*, and the definition of the restricting argument is also identical. Each caller is reported on its own line. The format differs slightly depending on the profiler that produced the stats:

- With *profile*, a number is shown in parentheses after each caller to show how many times this specific call was made. For convenience, a second non-parenthesized number repeats the cumulative time spent in the function at the right.
- With *cProfile*, each caller is preceded by three numbers: the number of times this specific call was made, and the total and cumulative times spent in the current function while it was invoked by this specific caller.

print_callees(**restrictions*)
This method for the *Stats* class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the *print_callers()* method.

28.4.5 What Is Deterministic Profiling?

Deterministic profiling is meant to reflect the fact that all *function call*, *function return*, and *exception* events are monitored, and precise timings are made for the intervals between these events (during which time the user's code is executing). In contrast, *statistical profiling* (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required to do deterministic profiling. Python automatically provides a *hook* (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify "hot loops" that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.

28.4.6 Limitations

One limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying "clock" is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate than the underlying clock. If enough measurements are taken, then the "error" will tend to average out. Unfortunately, removing this first error induces a second source of error.

The second problem is that it "takes a while" from when an event is dispatched until the profiler's call to get the time actually *gets* the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock's value was obtained (and then squirreled away), until the user's code is once again executing. As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (less than one clock tick), but it *can* accumulate and become very significant.

The problem is more important with `profile` than with the lower-overhead `cProfile`. For this reason, `profile` provides a means of calibrating itself for a given platform so that this error can be probabilistically (on the average) removed. After the profiler is calibrated, it will be more accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-).) Do *not* be alarmed by negative numbers in the profile. They should *only* appear if you have calibrated your profiler, and the results are actually better than without calibration.

28.4.7 Calibration

The profiler of the `profile` module subtracts a constant from each event handling time to compensate for the overhead of calling the time function, and socking away the results. By default, the constant is 0. The following procedure can be used to obtain a better constant for a given platform (see *Limitations*).

```
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on a 1.8Ghz Intel Core i5 running Mac OS X, and using Python's time.process_time() as the timer, the magical number is about 4.04e-6.

The object of this exercise is to get a fairly consistent result. If your computer is *very* fast, or your timer function has poor resolution, you might have to pass 100000, or even 1000000, to get consistent results.

When you have a consistent answer, there are three ways you can use it:

```
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
```

(continua na próxima página)

(continuação da página anterior)

```
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

If you have a choice, you are better off choosing a smaller constant, and then your results will "less often" show up as negative in profile statistics.

28.4.8 Using a custom timer

If you want to change how current time is determined (for example, to force use of wall-clock time or elapsed process time), pass the timing function you want to the `Profile` class constructor:

```
pr = profile.Profile(your_time_func)
```

The resulting profiler will then call `your_time_func`. Depending on whether you are using *profile.Profile* or `cProfile.Profile`, `your_time_func`'s return value will be interpreted differently:

profile.Profile `your_time_func` should return a single number, or a list of numbers whose sum is the current time (like what *os.times()* returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the profiler class for the timer function that you choose (see *Calibration*). For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (*os.times()* is *pretty* bad, as it returns a tuple of floating point values). If you want to substitute a better timer in the cleanest fashion, derive a class and hardwire a replacement dispatch method that best handles your timer call, along with the appropriate calibration constant.

cProfile.Profile `your_time_func` should return a single number. If it returns integers, you can also invoke the class constructor with a second argument specifying the real duration of one unit of time. For example, if `your_integer_time_func` returns times measured in thousands of seconds, you would construct the `Profile` instance as follows:

```
pr = cProfile.Profile(your_integer_time_func, 0.001)
```

As the `cProfile.Profile` class cannot be calibrated, custom timer functions should be used with care and should be as fast as possible. For the best results with a custom timer, it might be necessary to hard-code it in the C source of the internal `_lsprof` module.

Python 3.3 adds several new functions in *time* that can be used to make precise measurements of process or wall-clock time. For example, see *time.perf_counter()*.

28.5 `timeit` — Measure execution time of small code snippets

Código Fonte: Lib/timeit.py

This module provides a simple way to time small bits of Python code. It has both a *Interface de Linha de Comando* as well as a *callable* one. It avoids a number of common traps for measuring execution times. See also Tim Peters' introduction to the "Algorithms" chapter in the *Python Cookbook*, published by O'Reilly.

28.5.1 Exemplos básicos

The following example shows how the *Interface de Linha de Comando* can be used to compare three different expressions:

```
$ python3 -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 5: 30.2 usec per loop
$ python3 -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 5: 27.5 usec per loop
$ python3 -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 5: 23.2 usec per loop
```

This can be achieved from the *Python Interface* with:

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.3018611848820001
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.2727368790656328
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.23702679807320237
```

A callable can also be passed from the *Python Interface*:

```
>>> timeit.timeit(lambda: "-".join(map(str, range(100))), number=10000)
0.19665591977536678
```

Note however that `timeit()` will automatically determine the number of repetitions only when the command-line interface is used. In the *Exemplos* section you can find more advanced examples.

28.5.2 Python Interface

The module defines three convenience functions and a public class:

`timeit.`**`timeit`**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *number=1000000*, *globals=None*)
Create a `Timer` instance with the given statement, *setup* code and *timer* function and run its `timeit()` method with *number* executions. The optional *globals* argument specifies a namespace in which to execute the code.

Alterado na versão 3.5: The optional *globals* parameter was added.

`timeit.`**`repeat`**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *repeat=5*, *number=1000000*, *globals=None*)
Create a `Timer` instance with the given statement, *setup* code and *timer* function and run its `repeat()` method with the given *repeat* count and *number* executions. The optional *globals* argument specifies a namespace in which to execute the code.

Alterado na versão 3.5: The optional *globals* parameter was added.

Alterado na versão 3.7: Default value of *repeat* changed from 3 to 5.

`timeit.`**`default_timer`**()
The default timer, which is always `time.perf_counter()`.

Alterado na versão 3.3: `time.perf_counter()` is now the default timer.

`class` `timeit.`**`Timer`**(*stmt='pass'*, *setup='pass'*, *timer=<timer function>*, *globals=None*)
Class for timing execution speed of small code snippets.

The constructor takes a statement to be timed, an additional statement used for setup, and a timer function. Both statements default to `'pass'`; the timer function is platform-dependent (see the module doc string). *stmt* and

setup may also contain multiple statements separated by `;` or newlines, as long as they don't contain multi-line string literals. The statement will by default be executed within timeit's namespace; this behavior can be controlled by passing a namespace to *globals*.

To measure the execution time of the first statement, use the `timeit()` method. The `repeat()` and `autorange()` methods are convenience methods to call `timeit()` multiple times.

The execution time of *setup* is excluded from the overall timed execution run.

The *stmt* and *setup* parameters can also take objects that are callable without arguments. This will embed calls to them in a timer function that will then be executed by `timeit()`. Note that the timing overhead is a little larger in this case because of the extra function calls.

Alterado na versão 3.5: The optional *globals* parameter was added.

timeit(*number=1000000*)

Time *number* executions of the main statement. This executes the setup statement once, and then returns the time it takes to execute the main statement a number of times, measured in seconds as a float. The argument is the number of times through the loop, defaulting to one million. The main statement, the setup statement and the timer function to be used are passed to the constructor.

Nota: By default, `timeit()` temporarily turns off *garbage collection* during the timing. The advantage of this approach is that it makes independent timings more comparable. The disadvantage is that GC may be an important component of the performance of the function being measured. If so, GC can be re-enabled as the first statement in the *setup* string. For example:

```
timeit.Timer('for i in range(10): oct(i)', 'gc.enable()').timeit()
```

---

autorange(*callback=None*)

Automatically determine how many times to call `timeit()`.

This is a convenience function that calls `timeit()` repeatedly so that the total time >= 0.2 second, returning the eventual (number of loops, time taken for that number of loops). It calls `timeit()` with increasing numbers from the sequence 1, 2, 5, 10, 20, 50, … until the time taken is at least 0.2 second.

If *callback* is given and is not `None`, it will be called after each trial with two arguments: `callback(number, time_taken)`.

Novo na versão 3.6.

repeat(*repeat=5*, *number=1000000*)

Call `timeit()` a few times.

This is a convenience function that calls the `timeit()` repeatedly, returning a list of results. The first argument specifies how many times to call `timeit()`. The second argument specifies the *number* argument for `timeit()`.

Nota: It's tempting to calculate mean and standard deviation from the result vector and report these. However, this is not very useful. In a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python's speed, but by other processes interfering with your timing accuracy. So the `min()` of the result is probably the only number you should be interested in. After that, you should look at the entire vector and apply common sense rather than statistics.

Alterado na versão 3.7: Default value of *repeat* changed from 3 to 5.

print_exc(*file=None*)
Helper to print a traceback from the timed code.

Uso típico:

```
t = Timer(...)       # outside the try/except
try:
    t.timeit(...)    # or t.repeat(...)
except Exception:
    t.print_exc()
```

The advantage over the standard traceback is that source lines in the compiled template will be displayed. The optional *file* argument directs where the traceback is sent; it defaults to *sys.stderr*.

28.5.3 Interface de Linha de Comando

When called as a program from the command line, the following form is used:

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-h] [statement ...]
```

Where the following options are understood:

-n N, **--number**=N
how many times to execute 'statement'

-r N, **--repeat**=N
how many times to repeat the timer (default 5)

-s S, **--setup**=S
statement to be executed once initially (default `pass`)

-p, **--process**
measure process time, not wallclock time, using *time.process_time()* instead of *time.perf_counter()*, which is the default

Novo na versão 3.3.

-u, **--unit**=U

specify a time unit for timer output; can select nsec, usec, msec, or sec

Novo na versão 3.5.

-v, **--verbose**
print raw timing results; repeat for more digits precision

-h, **--help**
print a short usage message and exit

A multi-line statement may be given by specifying each line as a separate statement argument; indented lines are possible by enclosing an argument in quotes and using leading spaces. Multiple *-s* options are treated similarly.

If *-n* is not given, a suitable number of loops is calculated by trying increasing numbers from the sequence 1, 2, 5, 10, 20, 50, … until the total time is at least 0.2 seconds.

default_timer() measurements can be affected by other programs running on the same machine, so the best thing to do when accurate timing is necessary is to repeat the timing a few times and use the best time. The *-r* option is good for this; the default of 5 repetitions is probably enough in most cases. You can use *time.process_time()* to measure CPU time.

Nota: There is a certain baseline overhead associated with executing a pass statement. The code here doesn't try to hide it, but you should be aware of it. The baseline overhead can be measured by invoking the program without arguments, and it might differ between Python versions.

28.5.4 Exemplos

É possível fornecer uma instrução de configuração que é executada apenas uma vez no início:

```
$ python -m timeit -s 'text = "sample string"; char = "g"'  'char in text'
5000000 loops, best of 5: 0.0877 usec per loop
$ python -m timeit -s 'text = "sample string"; char = "g"'  'text.find(char)'
1000000 loops, best of 5: 0.342 usec per loop
```

```
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"; char = "g"')
1.7246671520006203
```

The same can be done using the *Timer* class and its methods:

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40183617287970225, 0.37027556854118704, 0.38344867356679524, 0.3712595970846668, 0.
↪37866875250654886]
```

The following examples show how to time expressions that contain multiple lines. Here we compare the cost of using *hasattr()* vs. `try`/`except` to test for missing and present object attributes:

```
$ python -m timeit 'try:' '  str.__bool__' 'except AttributeError:' '  pass'
20000 loops, best of 5: 15.7 usec per loop
$ python -m timeit 'if hasattr(str, "__bool__"): pass'
50000 loops, best of 5: 4.26 usec per loop

$ python -m timeit 'try:' '  int.__bool__' 'except AttributeError:' '  pass'
200000 loops, best of 5: 1.43 usec per loop
$ python -m timeit 'if hasattr(int, "__bool__"): pass'
100000 loops, best of 5: 2.23 usec per loop
```

```
>>> import timeit
>>> # attribute is missing
>>> s = """\
... try:
...     str.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
```

(continua na próxima página)

(continuação da página anterior)

```
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>>
>>> # attribute is present
>>> s = """\
... try:
...     int.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

To give the *timeit* module access to functions you define, you can pass a *setup* parameter which contains an import statement:

```
def test():
    """Stupid test function"""
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

Another option is to pass *globals()* to the *globals* parameter, which will cause the code to be executed within your current global namespace. This can be more convenient than individually specifying imports:

```
def f(x):
    return x**2
def g(x):
    return x**4
def h(x):
    return x**8

import timeit
print(timeit.timeit('[func(42) for func in (f,g,h)]', globals=globals()))
```

28.6 `trace` — Rastreia ou acompanha a execução de instruções Python

Código Fonte: Lib/trace.py

O módulo *trace* permite que você rastreie a execução do programa, gere listagens de cobertura de instrução anotada, imprima relações de chamador/receptor e funções de lista executadas durante a execução de um programa. Ele pode ser usado em outro programa ou na linha de comando.

Ver também:

Coverage.py Uma popular ferramenta de cobertura de terceiros que fornece saída HTML junto com recursos avançados, como cobertura de ramificações.

28.6.1 Uso da linha de comando

O módulo *trace* pode ser chamado a partir da linha de comando. Pode ser tão simples quanto:

```
python -m trace --count -C . somefile.py ...
```

O comando acima irá executar `algumarquivo.py` e gerar listagens anotadas de todos os módulos Python importados durante a execução para o diretório atual.

--help
Exibe o uso e sai.

--version
Exibe a versão do módulo e sai.

Opções principais

Pelo menos uma das seguintes opções deve ser especificada ao invocar *trace*. A opção *--listfuncs* é mutuamente exclusiva com as opções *--trace* e *--count*. Quando *--listfuncs* é fornecida, nem *--count* nem *--trace* são aceitas, e vice-versa.

-c, --count
Produz um conjunto de arquivos de listagem anotada após a conclusão do programa que mostra quantas vezes cada instrução foi executada. Veja também *--coverdir*, *--file* e *--no-report* abaixo.

-t, --trace
Exibe linhas como elas são executadas.

-l, --listfuncs
Exibe as funções executadas executando o programa.

-r, --report
Produz uma lista anotada de uma execução de programa anterior que usava a opção *--count* e *--file*. Isso não executa nenhum código.

-T, --trackcalls
Exibe os relacionamentos de chamada expostos ao executar o programa.

Modificadores

-f, --file=<file>
Nome de um arquivo para acumular contagens em várias execuções de rastreamento. Deve ser usado com a opção *--count*.

-C, --coverdir=<dir>
Diretório para onde vão os arquivos de relatório. O relatório de cobertura para `pacote.módulo` é escrito em arquivo *dir/pacote/módulo*`.cover`.

-m, --missing
Ao gerar listagens anotadas, marca as linhas que não foram executadas com >>>>>>.

-s, --summary
Ao usar *--count* ou *--report*, escreve um breve resumo no stdout para cada arquivo processado.

-R, --no-report
Não gera listagens anotadas. Isso é útil se você pretende fazer várias execuções com *--count*, e então produzir um único conjunto de listagens anotadas no final.

-g, --timing
Prefixa cada linha com o tempo desde o início do programa. Usado apenas durante o rastreamento.

Filtros

Essas opções podem ser repetidas várias vezes.

--ignore-module=<mod>
Ignora cada um dos nomes de módulo fornecidos e seus submódulos (se for um pacote). O argumento pode ser uma lista de nomes separados por uma vírgula.

--ignore-dir=<dir>
Ignora todos os módulos e pacotes no diretório e subdiretórios nomeados. O argumento pode ser uma lista de diretórios separados por *os.pathsep*.

28.6.2 Interface programática

class trace.**Trace**(*count=1*, *trace=1*, *countfuncs=0*, *countcallers=0*, *ignoremods=()*, *ignoredirs=()*, *infile=None*, *outfile=None*, *timing=False*)
Cria um objeto para rastrear a execução de uma única instrução ou expressão. Todos os parâmetros são opcionais. *count* ativa a contagem de números de linha. *trace* ativa o rastreamento de execução de linha. *countfuncs* ativa a listagem das funções chamadas durante a execução. *countcallers* ativa o rastreamento de relacionamento de chamada. *ignoremods* é uma lista de módulos ou pacotes a serem ignorados. *ignoreirs* é uma lista de diretórios cujos módulos ou pacotes devem ser ignorados. *infile* é o nome do arquivo do qual deve ler as informações de contagem armazenadas. *outfile* é o nome do arquivo no qual deve escrever as informações de contagem atualizadas. *timing* ativa a exibição de um carimbo de data/hora relativo ao momento em que o rastreamento foi iniciado.

run(*cmd*)
Executa o comando e reúne estatísticas da execução com os parâmetros de rastreamento atuais. *cmd* deve ser uma string ou objeto código, adequado para passar para *exec()*.

runctx(*cmd*, *globals=None*, *locals=None*)
Executa o comando e reúne estatísticas da execução com os parâmetros de rastreamento atuais, nos ambientes global e local definidos. Se não for definido, *globals* e *locals* usam como padrão dicionários vazios.

runfunc(*func*, **args*, ***kwds*)
Chama *func* com os argumentos fornecidos sob controle do objeto *Trace* com os parâmetros de rastreamento atuais.

results()
Retorna um objeto *CoverageResults* que contém os resultados cumulativos de todas as chamadas anteriores para `run`, `runctx` e `runfunc` para a instância *Trace* fornecida. Não redefine os resultados de rastreamento acumulados.

class trace.**CoverageResults**
Um contêiner para resultados de cobertura, criado por *Trace.results()*. Não deve ser criado diretamente pelo usuário.

update(*other*)
Mescla dados de outro objeto *CoverageResults*.

write_results(*show_missing=True*, *summary=False*, *coverdir=None*)
Escreve os resultados da cobertura. Defina *show_missing* para mostrar as linhas que não tiveram ocorrências. Defina o *summary* para incluir na saída o resumo da cobertura por módulo. *coverdir* especifica o diretório no qual os arquivos de resultados de cobertura serão enviados. Se for `None`, os resultados de cada arquivo de origem são colocados em seu diretório.

Um exemplo simples que demonstra o uso da interface programática:

```
import sys
import trace

# create a Trace object, telling it what to ignore, and whether to
# do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# run the new command using the given tracer
tracer.run('main()')

# make a report, placing output in the current directory
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```

28.7 `tracemalloc` — Trace memory allocations

Novo na versão 3.4.

Código Fonte: Lib/tracemalloc.py

The tracemalloc module is a debug tool to trace memory blocks allocated by Python. It provides the following information:

- Traceback where an object was allocated
- Statistics on allocated memory blocks per filename and per line number: total size, number and average size of allocated memory blocks
- Compute the differences between two snapshots to detect memory leaks

To trace most memory blocks allocated by Python, the module should be started as early as possible by setting the `PYTHONTRACEMALLOC` environment variable to `1`, or by using `-X tracemalloc` command line option. The *`tracemalloc.start()`* function can be called at runtime to start tracing Python memory allocations.

By default, a trace of an allocated memory block only stores the most recent frame (1 frame). To store 25 frames at startup: set the `PYTHONTRACEMALLOC` environment variable to `25`, or use the `-X tracemalloc=25` command line option.

28.7.1 Exemplos

Exibe o top 10

Display the 10 files allocating the most memory:

```
import tracemalloc

tracemalloc.start()

# ... run your application ...
```

(continua na próxima página)

(continuação da página anterior)

```
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output of the Python test suite:

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=39328, average=126 B
<frozen importlib._bootstrap>:284: size=521 KiB, count=3199, average=167 B
/usr/lib/python3.4/collections/__init__.py:368: size=244 KiB, count=2315, average=108␣
↪B
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB, count=779, average=243 B
/usr/lib/python3.4/unittest/case.py:402: size=154 KiB, count=378, average=416 B
/usr/lib/python3.4/abc.py:133: size=88.7 KiB, count=347, average=262 B
<frozen importlib._bootstrap>:1446: size=70.4 KiB, count=911, average=79 B
<frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25, average=2131 B
<string>:5: size=49.7 KiB, count=148, average=344 B
/usr/lib/python3.4/sysconfig.py:411: size=48.0 KiB, count=1, average=48.0 KiB
```

We can see that Python loaded `4855 KiB` data (bytecode and constants) from modules and that the *collections* module allocated `244 KiB` to build *namedtuple* types.

See *Snapshot.statistics()* for more options.

Compute differences

Take two snapshots and display the differences:

```
import tracemalloc
tracemalloc.start()
# ... start your application ...

snapshot1 = tracemalloc.take_snapshot()
# ... call the function leaking memory ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output before/after running some tests of the Python test suite:

```
[ Top 10 differences ]
<frozen importlib._bootstrap>:716: size=8173 KiB (+4428 KiB), count=71332 (+39369),␣
↪average=117 B
/usr/lib/python3.4/linecache.py:127: size=940 KiB (+940 KiB), count=8106 (+8106),␣
↪average=119 B
/usr/lib/python3.4/unittest/case.py:571: size=298 KiB (+298 KiB), count=589 (+589),␣
↪average=519 B
<frozen importlib._bootstrap>:284: size=1005 KiB (+166 KiB), count=7423 (+1526),␣
↪average=139 B
```

(continua na próxima página)

(continuação da página anterior)

```
/usr/lib/python3.4/mimetypes.py:217: size=112 KiB (+112 KiB), count=1334 (+1334),␣
↪average=86 B
/usr/lib/python3.4/http/server.py:848: size=96.0 KiB (+96.0 KiB), count=1 (+1),␣
↪average=96.0 KiB
/usr/lib/python3.4/inspect.py:1465: size=83.5 KiB (+83.5 KiB), count=109 (+109),␣
↪average=784 B
/usr/lib/python3.4/unittest/mock.py:491: size=77.7 KiB (+77.7 KiB), count=143 (+143),␣
↪average=557 B
/usr/lib/python3.4/urllib/parse.py:476: size=71.8 KiB (+71.8 KiB), count=969 (+969),␣
↪average=76 B
/usr/lib/python3.4/contextlib.py:38: size=67.2 KiB (+67.2 KiB), count=126 (+126),␣
↪average=546 B
```

We can see that Python has loaded `8173 KiB` of module data (bytecode and constants), and that this is `4428 KiB` more than had been loaded before the tests, when the previous snapshot was taken. Similarly, the *`linecache`* module has cached `940 KiB` of Python source code to format tracebacks, all of it since the previous snapshot.

If the system has little free memory, snapshots can be written on disk using the *`Snapshot.dump()`* method to analyze the snapshot offline. Then use the *`Snapshot.load()`* method reload the snapshot.

Get the traceback of a memory block

Code to display the traceback of the biggest memory block:

```
import tracemalloc

# Store 25 frames
tracemalloc.start(25)

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# pick the biggest memory block
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))
for line in stat.traceback.format():
    print(line)
```

Example of output of the Python test suite (traceback limited to 25 frames):

```
903 memory blocks: 870.1 KiB
  File "<frozen importlib._bootstrap>", line 716
  File "<frozen importlib._bootstrap>", line 1036
  File "<frozen importlib._bootstrap>", line 934
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/doctest.py", line 101
    import pdb
  File "<frozen importlib._bootstrap>", line 284
  File "<frozen importlib._bootstrap>", line 938
  File "<frozen importlib._bootstrap>", line 1068
```

(continua na próxima página)

(continuação da página anterior)

```
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/test/support/__init__.py", line 1728
    import doctest
  File "/usr/lib/python3.4/test/test_pickletools.py", line 21
    support.run_doctest(pickletools)
  File "/usr/lib/python3.4/test/regrtest.py", line 1276
    test_runner()
  File "/usr/lib/python3.4/test/regrtest.py", line 976
    display_failure=not verbose)
  File "/usr/lib/python3.4/test/regrtest.py", line 761
    match_tests=ns.match_tests)
  File "/usr/lib/python3.4/test/regrtest.py", line 1563
    main()
  File "/usr/lib/python3.4/test/__main__.py", line 3
    regrtest.main_in_temp_cwd()
  File "/usr/lib/python3.4/runpy.py", line 73
    exec(code, run_globals)
  File "/usr/lib/python3.4/runpy.py", line 160
    "__main__", fname, loader, pkg_name)
```

We can see that the most memory was allocated in the *importlib* module to load data (bytecode and constants) from modules: `870.1 KiB`. The traceback is where the *importlib* loaded data most recently: on the `import pdb` line of the *doctest* module. The traceback may change if a new module is loaded.

Pretty top

Code to display the 10 lines allocating the most memory with a pretty output, ignoring `<frozen importlib._bootstrap>` and `<unknown>` files:

```
import linecache
import os
import tracemalloc

def display_top(snapshot, key_type='lineno', limit=10):
    snapshot = snapshot.filter_traces((
        tracemalloc.Filter(False, "<frozen importlib._bootstrap>"),
        tracemalloc.Filter(False, "<unknown>"),
    ))
    top_stats = snapshot.statistics(key_type)

    print("Top %s lines" % limit)
    for index, stat in enumerate(top_stats[:limit], 1):
        frame = stat.traceback[0]
        print("#%s: %s:%s: %.1f KiB"
              % (index, frame.filename, frame.lineno, stat.size / 1024))
        line = linecache.getline(frame.filename, frame.lineno).strip()
        if line:
            print('    %s' % line)

    other = top_stats[limit:]
    if other:
        size = sum(stat.size for stat in other)
        print("%s other: %.1f KiB" % (len(other), size / 1024))
```

(continua na próxima página)

(continuação da página anterior)

```
    total = sum(stat.size for stat in top_stats)
    print("Total allocated size: %.1f KiB" % (total / 1024))

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)
```

Example of output of the Python test suite:

```
Top 10 lines
#1: Lib/base64.py:414: 419.8 KiB
    _b85chars2 = [(a + b) for a in _b85chars for b in _b85chars]
#2: Lib/base64.py:306: 419.8 KiB
    _a85chars2 = [(a + b) for a in _a85chars for b in _a85chars]
#3: collections/__init__.py:368: 293.6 KiB
    exec(class_definition, namespace)
#4: Lib/abc.py:133: 115.2 KiB
    cls = super().__new__(mcls, name, bases, namespace)
#5: unittest/case.py:574: 103.1 KiB
    testMethod()
#6: Lib/linecache.py:127: 95.4 KiB
    lines = fp.readlines()
#7: urllib/parse.py:476: 71.8 KiB
    for a in _hexdig for b in _hexdig}
#8: <string>:5: 62.0 KiB
#9: Lib/_weakrefset.py:37: 60.0 KiB
    self.data = set()
#10: Lib/base64.py:142: 59.8 KiB
    _b32tab2 = [a + b for a in _b32tab for b in _b32tab]
6220 other: 3602.8 KiB
Total allocated size: 5303.1 KiB
```

See *Snapshot.statistics()* for more options.

28.7.2 API

Funções

tracemalloc.**clear_traces**()
: Clear traces of memory blocks allocated by Python.

  See also *stop()*.

tracemalloc.**get_object_traceback**(*obj*)
: Get the traceback where the Python object *obj* was allocated. Return a *Traceback* instance, or `None` if the *tracemalloc* module is not tracing memory allocations or did not trace the allocation of the object.

  See also *gc.get_referrers()* and *sys.getsizeof()* functions.

tracemalloc.**get_traceback_limit**()
: Get the maximum number of frames stored in the traceback of a trace.

  The *tracemalloc* module must be tracing memory allocations to get the limit, otherwise an exception is raised.

The limit is set by the *start()* function.

tracemalloc.**get_traced_memory**()
Get the current size and peak size of memory blocks traced by the *tracemalloc* module as a tuple: `(current: int, peak: int)`.

tracemalloc.**get_tracemalloc_memory**()
Get the memory usage in bytes of the *tracemalloc* module used to store traces of memory blocks. Return an *int*.

tracemalloc.**is_tracing**()
`True` if the *tracemalloc* module is tracing Python memory allocations, `False` otherwise.

See also *start()* and *stop()* functions.

tracemalloc.**start**(*nframe: int=1*)
Start tracing Python memory allocations: install hooks on Python memory allocators. Collected tracebacks of traces will be limited to *nframe* frames. By default, a trace of a memory block only stores the most recent frame: the limit is `1`. *nframe* must be greater or equal to `1`.

Storing more than `1` frame is only useful to compute statistics grouped by `'traceback'` or to compute cumulative statistics: see the *Snapshot.compare_to()* and *Snapshot.statistics()* methods.

Storing more frames increases the memory and CPU overhead of the *tracemalloc* module. Use the *get_tracemalloc_memory()* function to measure how much memory is used by the *tracemalloc* module.

The `PYTHONTRACEMALLOC` environment variable (`PYTHONTRACEMALLOC=NFRAME`) and the `-X tracemalloc=NFRAME` command line option can be used to start tracing at startup.

See also *stop()*, *is_tracing()* and *get_traceback_limit()* functions.

tracemalloc.**stop**()
Stop tracing Python memory allocations: uninstall hooks on Python memory allocators. Also clears all previously collected traces of memory blocks allocated by Python.

Call *take_snapshot()* function to take a snapshot of traces before clearing them.

See also *start()*, *is_tracing()* and *clear_traces()* functions.

tracemalloc.**take_snapshot**()
Take a snapshot of traces of memory blocks allocated by Python. Return a new *Snapshot* instance.

The snapshot does not include memory blocks allocated before the *tracemalloc* module started to trace memory allocations.

Tracebacks of traces are limited to *get_traceback_limit()* frames. Use the *nframe* parameter of the *start()* function to store more frames.

The *tracemalloc* module must be tracing memory allocations to take a snapshot, see the *start()* function.

See also the *get_object_traceback()* function.

DomainFilter

class tracemalloc.**DomainFilter**(*inclusive: bool*, *domain: int*)
Filter traces of memory blocks by their address space (domain).

Novo na versão 3.6.

inclusive
If *inclusive* is `True` (include), match memory blocks allocated in the address space *domain*.

If *inclusive* is `False` (exclude), match memory blocks not allocated in the address space *domain*.

domain
Address space of a memory block (`int`). Read-only property.

Filter

class tracemalloc.**Filter**(*inclusive: bool*, *filename_pattern: str*, *lineno: int=None*, *all_frames: bool=False*, *domain: int=None*)
Filter on traces of memory blocks.

See the *fnmatch.fnmatch()* function for the syntax of *filename_pattern*. The `'.pyc'` file extension is replaced with `'.py'`.

Exemplos:

- `Filter(True, subprocess.__file__)` only includes traces of the *subprocess* module
- `Filter(False, tracemalloc.__file__)` excludes traces of the *tracemalloc* module
- `Filter(False, "<unknown>")` excludes empty tracebacks

Alterado na versão 3.5: The `'.pyo'` file extension is no longer replaced with `'.py'`.

Alterado na versão 3.6: Added the *domain* attribute.

domain
Address space of a memory block (`int` or `None`).

tracemalloc uses the domain `0` to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

inclusive
If *inclusive* is `True` (include), only match memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

If *inclusive* is `False` (exclude), ignore memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

lineno
Line number (`int`) of the filter. If *lineno* is `None`, the filter matches any line number.

filename_pattern
Filename pattern of the filter (`str`). Read-only property.

all_frames
If *all_frames* is `True`, all frames of the traceback are checked. If *all_frames* is `False`, only the most recent frame is checked.

This attribute has no effect if the traceback limit is `1`. See the *get_traceback_limit()* function and *Snapshot.traceback_limit* attribute.

Frame

class tracemalloc.**Frame**
Frame de um Traceback

The *Traceback* class is a sequence of *Frame* instances.

filename
Filename (str).

lineno
Número da linha (int).

Snapshot

class tracemalloc.**Snapshot**
Snapshot of traces of memory blocks allocated by Python.

The *take_snapshot()* function creates a snapshot instance.

compare_to(*old_snapshot: Snapshot*, *key_type: str*, *cumulative: bool=False*)
Compute the differences with an old snapshot. Get statistics as a sorted list of *StatisticDiff* instances grouped by *key_type*.

See the *Snapshot.statistics()* method for *key_type* and *cumulative* parameters.

The result is sorted from the biggest to the smallest by: absolute value of *StatisticDiff.size_diff*, *StatisticDiff.size*, absolute value of *StatisticDiff.count_diff*, *Statistic.count* and then by *StatisticDiff.traceback*.

dump(*filename*)
Write the snapshot into a file.

Use *load()* to reload the snapshot.

filter_traces(*filters*)
Create a new *Snapshot* instance with a filtered *traces* sequence, *filters* is a list of *DomainFilter* and *Filter* instances. If *filters* is an empty list, return a new *Snapshot* instance with a copy of the traces.

All inclusive filters are applied at once, a trace is ignored if no inclusive filters match it. A trace is ignored if at least one exclusive filter matches it.

Alterado na versão 3.6: *DomainFilter* instances are now also accepted in *filters*.

classmethod load(*filename*)
Load a snapshot from a file.

See also *dump()*.

statistics(*key_type: str*, *cumulative: bool=False*)
Get statistics as a sorted list of *Statistic* instances grouped by *key_type*:

key_type	description
'filename'	filename
'lineno'	filename and line number
'traceback'	traceback

If *cumulative* is True, cumulate size and count of memory blocks of all frames of the traceback of a trace, not only the most recent frame. The cumulative mode can only be used with *key_type* equals to 'filename' and 'lineno'.

The result is sorted from the biggest to the smallest by: *Statistic.size*, *Statistic.count* and then by *Statistic.traceback*.

traceback_limit
Maximum number of frames stored in the traceback of *traces*: result of the *get_traceback_limit()* when the snapshot was taken.

traces
Traces of all memory blocks allocated by Python: sequence of *Trace* instances.

The sequence has an undefined order. Use the *Snapshot.statistics()* method to get a sorted list of statistics.

Statistic

class tracemalloc.**Statistic**
Statistic on memory allocations.

Snapshot.statistics() returns a list of *Statistic* instances.

See also the *StatisticDiff* class.

count
Number of memory blocks (`int`).

size
Total size of memory blocks in bytes (`int`).

traceback
Traceback where the memory block was allocated, *Traceback* instance.

StatisticDiff

class tracemalloc.**StatisticDiff**
Statistic difference on memory allocations between an old and a new *Snapshot* instance.

Snapshot.compare_to() returns a list of *StatisticDiff* instances. See also the *Statistic* class.

count
Number of memory blocks in the new snapshot (`int`): `0` if the memory blocks have been released in the new snapshot.

count_diff
Difference of number of memory blocks between the old and the new snapshots (`int`): `0` if the memory blocks have been allocated in the new snapshot.

size
Total size of memory blocks in bytes in the new snapshot (`int`): `0` if the memory blocks have been released in the new snapshot.

size_diff
Difference of total size of memory blocks in bytes between the old and the new snapshots (`int`): `0` if the memory blocks have been allocated in the new snapshot.

traceback
Traceback where the memory blocks were allocated, *Traceback* instance.

Trace

class tracemalloc.**Trace**
Trace of a memory block.

The *Snapshot.traces* attribute is a sequence of *Trace* instances.

Alterado na versão 3.6: Added the *domain* attribute.

domain
Address space of a memory block (`int`). Read-only property.

tracemalloc uses the domain `0` to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

size
Size of the memory block in bytes (`int`).

traceback
Traceback where the memory block was allocated, *Traceback* instance.

Traceback

class tracemalloc.**Traceback**
Sequence of *Frame* instances sorted from the oldest frame to the most recent frame.

A traceback contains at least `1` frame. If the `tracemalloc` module failed to get a frame, the filename `"<unknown>"` at line number `0` is used.

When a snapshot is taken, tracebacks of traces are limited to *get_traceback_limit()* frames. See the *take_snapshot()* function.

The *Trace.traceback* attribute is an instance of *Traceback* instance.

Alterado na versão 3.7: Frames are now sorted from the oldest to the most recent, instead of most recent to oldest.

format(*limit=None*, *most_recent_first=False*)
Format the traceback as a list of lines with newlines. Use the *linecache* module to retrieve lines from the source code. If *limit* is set, format the *limit* most recent frames if *limit* is positive. Otherwise, format the `abs(limit)` oldest frames. If *most_recent_first* is `True`, the order of the formatted frames is reversed, returning the most recent frame first instead of last.

Similar to the *traceback.format_tb()* function, except that *format()* does not include newlines.

Exemplo:

```
print("Traceback (most recent call first):")
for line in traceback:
    print(line)
```

Saída:

```
Traceback (most recent call first):
  File "test.py", line 9
    obj = Object()
  File "test.py", line 12
    tb = tracemalloc.get_object_traceback(f())
```

CAPÍTULO 29

Empacotamento e Distribuição de Software

Essas bibliotecas ajudam você a publicar e instalar software do Python. Embora esses módulos sejam projetados para funcionar em conjunto com o Python Package Index, eles também podem ser usados com um servidor de indexação local ou sem qualquer servidor de indexação.

29.1 `distutils` — Compilação e instalação de módulos do Python

O pacote *`distutils`* fornece suporte para criar e instalar módulos adicionais em uma instalação do Python. Os novos módulos podem ser um Python 100% puro, podem ser módulos de extensão escritos em C ou podem ser coleções de pacotes Python que incluem módulos codificados em Python e C.

A maioria dos usuários do Python *não* deseja usar esse módulo diretamente, mas usa as ferramentas de versão cruzada mantidas pela Python Packaging Authority. Em particular, setuptools é uma alternativa aprimorada para o *`distutils`* que fornece:

- suporte para declaração de dependências do projeto
- mecanismos adicionais para configurar quais arquivos devem ser incluídos em lançamentos de fonte (incluindo plugins para integração com sistemas de controle de versão)
- a capacidade de declarar "pontos de entrada" do projeto, os quais podem ser usados como base para sistemas de plugin do aplicativo.
- a capacidade para gerar automaticamente executáveis de linha de comando do Windows em tempo de instalação em vez de precisar de reconstruí-los
- comportamento consistente em todas as versões suportadas do Python

O instalador pip recomendado executa todos os scripts `setup.py` com `setuptools`, mesmo que o próprio script importe apenas `distutils`. Consulte o Guia do Usuário de Pacotes Python para obter mais informações.

Para os benefícios dos autores e usuários da ferramenta de empacotamento que buscam uma compreensão mais profunda dos detalhes do atual sistema de empacotamento e distribuição, a documentação legada baseada no *`distutils`* e a referência de API permanecem disponíveis:

- install-index
- distutils-index

29.2 ensurepip — Inicialização do instalador pip

Novo na versão 3.4.

O pacote *ensurepip* fornece suporte a fazer bootstrapping (isto é, inicializar) o instalador pip em uma instalação existente do Python ou em um ambiente virtual. Essa abordagem de bootstrapping reflete o fato de que pip é um projeto independente com seu próprio ciclo de lançamento, e a última versão estável disponível é fornecida com manutenção e lançamentos de recursos do interpretador de referência CPython.

Na maioria dos casos, os usuários finais do Python não precisam chamar esse módulo diretamente (como pip deve ser inicializado por padrão), mas pode ser necessário se a instalação do pip foi ignorada ao instalar o Python (ou ao criar um ambiente virtual) ou após desinstalar explicitamente pip.

Nota: Este módulo *não* acessa a Internet. Todos os componentes necessários para iniciar o pip estão incluídos como partes internas do pacote.

Ver também:

installing-index O guia do usuário final para instalar pacotes Python

PEP 453: Inicialização explícita de pip em instalações Python A justificativa e especificação originais para este módulo.

29.2.1 Interface de linha de comando

A interface da linha de comandos é chamada usando a opção -m do interpretador.

A invocação mais simples possível é:

```
python -m ensurepip
```

Essa invocação instalará pip se ainda não estiver instalada, mas, caso contrário, não fará nada. Para garantir que a versão instalada do pip seja pelo menos tão recente quanto a que acompanha o pacote ensurepip, passe a opção --upgrade:

```
python -m ensurepip --upgrade
```

Por padrão, pip é instalado no ambiente virtual atual (se houver um ativo) ou nos pacotes de sites do sistema (se não houver um ambiente virtual ativo). O local da instalação pode ser controlado através de duas opções adicionais de linha de comando:

- --root <dir>: Instala pip em relação ao diretório raiz fornecido, em vez da raiz do ambiente virtual atualmente ativo (se houver) ou a raiz padrão da instalação atual do Python.
- --user: Instala pip no diretório de pacotes do site do usuário em vez de globalmente para a instalação atual do Python (essa opção não é permitida dentro de um ambiente virtual ativo).

Por padrão, os scripts pipX e pipX.Y serão instalados (onde X.Y representa a versão do Python usada para invocar ensurepip). Os scripts instalados podem ser controlados através de duas opções adicionais de linha de comando:

- `--altinstall`: se uma instalação alternativa for solicitada, o script `pipX` *não* será instalado.
- `--default-pip`: se uma instalação "pip padrão" for solicitada, o script `pip` será instalado junto com os dois scripts comuns.

Fornecer as duas opções de seleção de script acionará uma exceção.

29.2.2 API do módulo

ensurepip expõe duas funções para uso programático:

ensurepip.**version**()
Retorna uma sequência que especifica a versão em pacote do pip que será instalado ao inicializar um ambiente.

ensurepip.**bootstrap**(*root=None*, *upgrade=False*, *user=False*, *altinstall=False*, *default_pip=False*, *verbosity=0*)
Inicializa `pip` no ambiente atual ou designado.

root especifica um diretório raiz alternativo para instalar em relação a. Se *root* for `None`, a instalação utilizará o local de instalação padrão para o ambiente atual.

upgrade indica se deve ou não atualizar uma instalação existente de uma versão anterior do `pip` para a versão empacotada.

user indica se é necessário usar o esquema do usuário em vez de instalar globalmente.

Por padrão, os scripts `pipX` e `pipX.Y` serão instalados (onde X.Y significa a versão atual do Python).

Se *altinstall* estiver definido, o `pipX` *não* será instalado.

Se *default_pip* estiver definido, o `pip` será instalado além dos dois scripts comuns.

Definir *altinstall* e *default_pip* acionará *ValueError*.

verbosity controla o nível de saída para *sys.stdout* da operação de inicialização.

Nota: O processo de inicialização tem efeitos colaterais em `sys.path` e `os.environ`. Invocar a interface da linha de comandos em um subprocesso permite que esses efeitos colaterais sejam evitados.

Nota: O processo de inicialização pode instalar módulos adicionais exigidos pelo `pip`, mas outro software não deve assumir que essas dependências sempre estarão presentes por padrão (como as dependências podem ser removidas em uma versão futura do `pip`).

29.3 venv— Criação de ambientes virtuais

Novo na versão 3.3.

Código-fonte: Lib/venv/

O módulo *venv* fornece suporte para a criação de "ambientes virtuais" leves com seus próprios diretórios de site, opcionalmente isolados dos diretórios de site do sistema. Cada ambiente virtual possui seu próprio binário Python (que corresponde à versão do binário usado para criar esse ambiente) e pode ter seu próprio conjunto independente de pacotes Python instalados nos diretórios do site.

Veja **PEP 405** para mais informações sonre Ambientes virtuais em Python.

Ver também:

Python Packaging User Guide: Creating and using virtual environments

Nota: The `pyvenv` script has been deprecated as of Python 3.6 in favor of using `python3 -m venv` to help prevent any potential confusion as to which Python interpreter a virtual environment will be based on.

29.3.1 Criando ambientes virtuais

A criação de *ambientes virtuais* é feita executando o comando `venv`:

```
python3 -m venv /path/to/new/virtual/environment
```

Running this command creates the target directory (creating any parent directories that don't exist already) and places a `pyvenv.cfg` file in it with a `home` key pointing to the Python installation from which the command was run. It also creates a `bin` (or `Scripts` on Windows) subdirectory containing a copy/symlink of the Python binary/binaries (as appropriate for the platform or arguments used at environment creation time). It also creates an (initially empty) `lib/pythonX.Y/site-packages` subdirectory (on Windows, this is `Lib\site-packages`). If an existing directory is specified, it will be re-used.

Obsoleto desde a versão 3.6: `pyvenv` era a ferramenta recomendada para criar ambientes virtuais para Python 3.3 e 3.4, e foi descontinuado no Python 3.6.

Alterado na versão 3.5: O uso de `venv` agora é recomendado para a criação de ambientes virtuais.

No Windows, invoque o comando `venv` da seguinte forma:

```
c:\>c:\Python35\python -m venv c:\path\to\myenv
```

Como alternativa, se você configurou as variáveis `PATH` e `PATHEXT` para a sua instalação do Python:

```
c:\>python -m venv c:\path\to\myenv
```

O comando, se executado com `-h`, mostrará as opções disponíveis:

```
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
            [--upgrade] [--without-pip] [--prompt PROMPT]
            ENV_DIR [ENV_DIR ...]

Creates virtual Python environments in one or more target directories.

positional arguments:
  ENV_DIR               A directory to create the environment in.

optional arguments:
  -h, --help            show this help message and exit
  --system-site-packages
                        Give the virtual environment access to the system
                        site-packages dir.
  --symlinks            Try to use symlinks rather than copies, when symlinks
                        are not the default for the platform.
  --copies              Try to use copies rather than symlinks, even when
                        symlinks are the default for the platform.
  --clear               Delete the contents of the environment directory if it
```

(continua na próxima página)

(continuação da página anterior)

```
                        already exists, before environment creation.
  --upgrade             Upgrade the environment directory to use this version
                        of Python, assuming Python has been upgraded in-place.
  --without-pip         Skips installing or upgrading pip in the virtual
                        environment (pip is bootstrapped by default)
  --prompt PROMPT       Provides an alternative prompt prefix for this
                        environment.

Once an environment has been created, you may wish to activate it, e.g. by
sourcing an activate script in its bin directory.
```

Alterado na versão 3.4: Instala o pip por padrão, adicionadas as opções `--without-pip` e `--copies`.

Alterado na versão 3.4: Nas versões anteriores, se o diretório de destino já existia, era levantado um erro, a menos que a opção `--clear` ou `--upgrade` fosse fornecida.

Nota: Embora haja suporte a links simbólicos no Windows, eles não são recomendados. É importante notar que clicar duas vezes em `python.exe` no Explorador de Arquivos resolverá o link simbólico com entusiasmo e ignorará o ambiente virtual.

O arquivo `pyvenv.cfg` criado também inclui a chave `include-system-site-packages`, definida como `true` se `venv` for executado com a opção `--system-site-packages`; caso contrário, `false`.

A menos que a opção `--without-pip` seja dada, *ensurepip* será chamado para inicializar o `pip` no ambiente virtual.

Vários caminhos podem ser dados para `venv`, caso em que um ambiente virtual idêntico será criado, de acordo com as opções fornecidas, em cada caminho fornecido.

Depois que um ambiente virtual é criado, ele pode ser "ativado" usando um script no diretório binário do ambiente virtual. A chamada do script é específica da plataforma (<*venv*> deve ser substituído pelo caminho do diretório que contém o ambiente virtual):

Plataforma	Shell	Comando para ativar o ambiente virtual
Posix	bash/zsh	$ source <venv>/bin/activate
	fish	$. <venv>/bin/activate.fish
	csh/tcsh	$ source <venv>/bin/activate.csh
Windows	cmd.exe	C:\> <venv>\Scripts\activate.bat
	PowerShell	PS C:\> <venv>\Scripts\Activate.ps1

Você não *precisa* especificamente ativar um ambiente; a ativação apenas anexa o diretório binário do ambiente virtual ao seu caminho, para que "python" invoque o interpretador Python do ambiente virtual e você possa executar scripts instalados sem precisar usar o caminho completo. No entanto, todos os scripts instalados em um ambiente virtual devem ser executáveis sem ativá-lo e executados com o Python do ambiente virtual automaticamente.

Você pode desativar um ambiente virtual digitando "deactivate" no seu shell. O mecanismo exato é específico da plataforma e é um detalhe interno da implementação (normalmente, uma função de script ou shell será usada).

Novo na versão 3.4: Scripts de ativação de `fish` e `csh`.

Nota: Um ambiente virtual é um ambiente Python, de modo que o interpretador, as bibliotecas e os scripts instalados nele são isolados daqueles instalados em outros ambientes virtuais e (por padrão) quaisquer bibliotecas instaladas em um Python do "sistema", ou seja, instalado como parte do seu sistema operacional.

Um ambiente virtual é uma árvore de diretórios que contém arquivos executáveis em Python e outros arquivos que indicam que é um ambiente virtual.

Ferramentas de instalação comuns, como setuptools e pip, funcionam conforme o esperado em ambientes virtuais. Em outras palavras, quando um ambiente virtual está ativo, eles instalam pacotes Python no ambiente virtual sem a necessidade de instruções explícitas.

Quando um ambiente virtual está ativo (ou seja, o interpretador Python do ambiente virtual está em execução), os atributos *sys.prefix* e *sys.exec_prefix* apontam para o diretório base do ambiente virtual, enquanto *sys.base_prefix* e *sys.base_exec_prefix* apontam para a instalação Python do ambiente não virtual que foi usada para criar o ambiente virtual. Se um ambiente virtual não estiver ativo, então *sys.prefix* é o mesmo que *sys.base_prefix* e *sys.exec_prefix* é o mesmo que *sys.base_exec_prefix* (todos eles apontam para uma instalação Python de ambiente não virtual).

Quando um ambiente virtual está ativo, todas as opções que alteram o caminho da instalação serão ignoradas em todos os arquivos de configuração *distutils* para impedir que projetos sejam inadvertidamente instalados fora do ambiente virtual.

Ao trabalhar em um shell de comando, os usuários podem ativar um ambiente virtual executando um script `activate` no diretório de executáveis do ambiente virtual (o nome do arquivo e o comando precisos para usar o arquivo dependem do shell), que precede o ambiente virtual diretório para executáveis na variável de ambiente `PATH` do shell em execução. Em outras circunstâncias, não há necessidade de ativar um ambiente virtual; scripts instalados em ambientes virtuais têm uma linha "shebang" que aponta para o interpretador Python do ambiente virtual. Isso significa que o script será executado com esse interpretador, independentemente do valor de `PATH`. No Windows, o processamento da linha "shebang" é suportado se você tiver o Python Launcher for Windows instalado (foi adicionado ao Python no 3.3 - consulte **PEP 397** para obter mais detalhes). Portanto, clicar duas vezes em um script instalado em uma janela do Explorador do Windows deve executar o script com o interpretador correto, sem que seja necessário fazer referência ao seu ambiente virtual em `PATH`.

29.3.2 API

O método de alto nível descrito acima utiliza uma API simples que fornece mecanismos para que criadores de ambientes virtuais de terceiros personalizem a criação do ambiente de acordo com suas necessidades, a classe *EnvBuilder*.

class venv.**EnvBuilder**(*system_site_packages=False*, *clear=False*, *symlinks=False*, *upgrade=False*, *with_pip=False*, *prompt=None*)

A classe *EnvBuilder* aceita os seguintes argumentos nomeados na instanciação:

- `system_site_packages` – um valor booleano indicando que os pacotes de sites do sistema Python devem estar disponíveis para o ambiente (o padrão é `False`).
- `clear` – um valor booleano que, se verdadeiro, excluirá o conteúdo de qualquer diretório de destino existente, antes de criar o ambiente.
- `symlinks` – um valor booleano que indica se você deseja vincular o binário Python ao invés de copiar.
- `upgrade` – um valor booleano que, se verdadeiro, atualizará um ambiente existente com o Python em execução - para uso quando o Python tiver sido atualizado no local (o padrão é `False`).
- `with_pip` – um valor booleano que, se verdadeiro, garante que o pip seja instalado no ambiente virtual. Isso usa *ensurepip* com a opção `--default-pip`.
- `prompt` – uma String a ser usada após o ambiente virtual ser ativado (o padrão é `None`, o que significa que o nome do diretório do ambiente seria usado).

Alterado na versão 3.4: Adicionado o parâmetro `with_pip`

Novo na versão 3.6: Adicionado o parâmetro `prompt`

Os criadores de ferramentas de ambiente virtual de terceiros estarão livres para usar a classe fornecida *EnvBuilder* como uma classe base.

O env-builder retornado é um objeto que possui um método, `create`:

create(*env_dir*)
Cria um ambiente virtual especificando o diretório de destino (absoluto ou relativo ao diretório atual) que deve conter o ambiente virtual. O método `create` cria o ambiente no diretório especificado ou levanta uma exceção apropriada.

O método `create` da classe *EnvBuilder* ilustra os ganchos disponíveis para personalização de subclasses:

```
def create(self, env_dir):
    """
    Create a virtualized Python environment in a directory.
    env_dir is the target directory to create an environment in.
    """
    env_dir = os.path.abspath(env_dir)
    context = self.ensure_directories(env_dir)
    self.create_configuration(context)
    self.setup_python(context)
    self.setup_scripts(context)
    self.post_setup(context)
```

Cada um dos métodos *ensure_directories()*, *create_configuration()*, *setup_python()*, *setup_scripts()* e *post_setup()* pode ser substituído.

ensure_directories(*env_dir*)
Cria o diretório do ambiente e todos os diretórios necessários e retorna um objeto de contexto. Este é apenas um suporte para atributos (como caminhos), para uso pelos outros métodos. Os diretórios já podem existir, desde que `clear` ou `upgrade` tenham sido especificados para permitir a operação em um diretório de ambiente existente.

create_configuration(*context*)
Cria o arquivo de configuração `pyvenv.cfg` no ambiente.

setup_python(*context*)
Cria uma cópia ou link simbólico para o executável Python no ambiente. Nos sistemas POSIX, se um executável específico `python3.x` foi usado, links simbólicos para `python` e `python3` serão criados apontando para esse executável, a menos que já existam arquivos com esses nomes.

setup_scripts(*context*)
Instala scripts de ativação apropriados para a plataforma no ambiente virtual.

post_setup(*context*)
Um método de espaço reservado que pode ser substituído em implementações de terceiros para pré-instalar pacotes no ambiente virtual ou executar outras etapas pós-criação.

Alterado na versão 3.7.2: O Windows agora usa scripts redirecionadores para `python[w].exe` em vez de copiar os binários reais. No 3.7.2, somente *setup_python()* não faz nada a menos que seja executado a partir de uma construção na árvore de origem.

Alterado na versão 3.7.3: O Windows copia os scripts redirecionadores como parte do *setup_python()* em vez de *setup_scripts()*. Este não foi o caso em 3.7.2. Ao usar links simbólicos, os executáveis originais serão vinculados.

Além disso, *EnvBuilder* fornece este método utilitário que pode ser chamado de *setup_scripts()* ou *post_setup()* nas subclasses para ajudar na instalação de scripts personalizados no ambiente virtual.

install_scripts(*context*, *path*)
path é o caminho para um diretório que deve conter subdiretórios “common”, “posix” e “nt”, cada um con-

tendo scripts destinados ao diretório bin no ambiente. O conteúdo de "common" e o diretório correspondente a *os.name* são copiados após alguma substituição de texto dos espaços reservados:

- __VENV_DIR__ é substituído pelo caminho absoluto do diretório do ambiente.
- __VENV_NAME__ é substituído pelo nome do ambiente (segmento do caminho final do diretório do ambiente).
- __VENV_PROMPT__ é substituído pelo prompt (o nome do ambiente entre parênteses e com o seguinte espaço)
- __VENV_BIN_NAME__ é substituído pelo nome do diretório bin (`bin` ou `Scripts`).
- __VENV_PYTHON__ é substituído pelo caminho absoluto do executável do ambiente.

É permitido que os diretórios existam (para quando um ambiente existente estiver sendo atualizado).

Há também uma função de conveniência no nível do módulo:

venv.**create**(*env_dir*, *system_site_packages=False*, *clear=False*, *symlinks=False*, *with_pip=False*, *prompt=None*)

Cria um *EnvBuilder* com os argumentos nomeados fornecidos e chame seu método *create()* com o argumento *env_dir*.

Novo na versão 3.3.

Alterado na versão 3.4: Adicionado o parâmetro `with_pip`

Alterado na versão 3.6: Adicionado o parâmetro `prompt`

29.3.3 Um exemplo de extensão de `EnvBuilder`

O script a seguir mostra como estender *EnvBuilder* implementando uma subclasse que instala setuptools e pip em um ambiente virtual criado:

```
import os
import os.path
from subprocess import Popen, PIPE
import sys
from threading import Thread
from urllib.parse import urlparse
from urllib.request import urlretrieve
import venv

class ExtendedEnvBuilder(venv.EnvBuilder):
    """
    This builder installs setuptools and pip so that you can pip or
    easy_install other packages into the created virtual environment.

    :param nodist: If true, setuptools and pip are not installed into the
                   created virtual environment.
    :param nopip: If true, pip is not installed into the created
                  virtual environment.
    :param progress: If setuptools or pip are installed, the progress of the
                     installation can be monitored by passing a progress
                     callable. If specified, it is called with two
                     arguments: a string indicating some progress, and a
                     context indicating where the string is coming from.
                     The context argument can have one of three values:
                     'main', indicating that it is called from virtualize()
```

(continua na próxima página)

(continuação da página anterior)

```
                    itself, and 'stdout' and 'stderr', which are obtained
                    by reading lines from the output streams of a subprocess
                    which is used to install the app.

                    If a callable is not specified, default progress
                    information is output to sys.stderr.
    """

    def __init__(self, *args, **kwargs):
        self.nodist = kwargs.pop('nodist', False)
        self.nopip = kwargs.pop('nopip', False)
        self.progress = kwargs.pop('progress', None)
        self.verbose = kwargs.pop('verbose', False)
        super().__init__(*args, **kwargs)

    def post_setup(self, context):
        """
        Set up any packages which need to be pre-installed into the
        virtual environment being created.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        os.environ['VIRTUAL_ENV'] = context.env_dir
        if not self.nodist:
            self.install_setuptools(context)
        # Can't install pip without setuptools
        if not self.nopip and not self.nodist:
            self.install_pip(context)

    def reader(self, stream, context):
        """
        Read lines from a subprocess' output stream and either pass to a progress
        callable (if specified) or write progress information to sys.stderr.
        """
        progress = self.progress
        while True:
            s = stream.readline()
            if not s:
                break
            if progress is not None:
                progress(s, context)
            else:
                if not self.verbose:
                    sys.stderr.write('.')
                else:
                    sys.stderr.write(s.decode('utf-8'))
                sys.stderr.flush()
        stream.close()

    def install_script(self, context, name, url):
        _, _, path, _, _, _ = urlparse(url)
        fn = os.path.split(path)[-1]
        binpath = context.bin_path
        distpath = os.path.join(binpath, fn)
        # Download script into the virtual environment's binaries folder
        urlretrieve(url, distpath)
```

(continua na próxima página)

(continuação da página anterior)

```
        progress = self.progress
        if self.verbose:
            term = '\n'
        else:
            term = ''
        if progress is not None:
            progress('Installing %s ...%s' % (name, term), 'main')
        else:
            sys.stderr.write('Installing %s ...%s' % (name, term))
            sys.stderr.flush()
        # Install in the virtual environment
        args = [context.env_exe, fn]
        p = Popen(args, stdout=PIPE, stderr=PIPE, cwd=binpath)
        t1 = Thread(target=self.reader, args=(p.stdout, 'stdout'))
        t1.start()
        t2 = Thread(target=self.reader, args=(p.stderr, 'stderr'))
        t2.start()
        p.wait()
        t1.join()
        t2.join()
        if progress is not None:
            progress('done.', 'main')
        else:
            sys.stderr.write('done.\n')
        # Clean up - no longer needed
        os.unlink(distpath)

    def install_setuptools(self, context):
        """
        Install setuptools in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = 'https://bitbucket.org/pypa/setuptools/downloads/ez_setup.py'
        self.install_script(context, 'setuptools', url)
        # clear up the setuptools archive which gets downloaded
        pred = lambda o: o.startswith('setuptools-') and o.endswith('.tar.gz')
        files = filter(pred, os.listdir(context.bin_path))
        for f in files:
            f = os.path.join(context.bin_path, f)
            os.unlink(f)

    def install_pip(self, context):
        """
        Install pip in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = 'https://raw.github.com/pypa/pip/master/contrib/get-pip.py'
        self.install_script(context, 'pip', url)

def main(args=None):
    compatible = True
    if sys.version_info < (3, 3):
        compatible = False
```

(continua na próxima página)

(continuação da página anterior)

```
    elif not hasattr(sys, 'base_prefix'):
        compatible = False
    if not compatible:
        raise ValueError('This script is only for use with '
                         'Python 3.3 or later')
    else:
        import argparse

        parser = argparse.ArgumentParser(prog=__name__,
                                         description='Creates virtual Python '
                                                     'environments in one or '
                                                     'more target '
                                                     'directories.')
        parser.add_argument('dirs', metavar='ENV_DIR', nargs='+',
                            help='A directory in which to create the
                                 'virtual environment.')
        parser.add_argument('--no-setuptools', default=False,
                            action='store_true', dest='nodist',
                            help="Don't install setuptools or pip in the "
                                 "virtual environment.")
        parser.add_argument('--no-pip', default=False,
                            action='store_true', dest='nopip',
                            help="Don't install pip in the virtual "
                                 "environment.")
        parser.add_argument('--system-site-packages', default=False,
                            action='store_true', dest='system_site',
                            help='Give the virtual environment access to the '
                                 'system site-packages dir.')
        if os.name == 'nt':
            use_symlinks = False
        else:
            use_symlinks = True
        parser.add_argument('--symlinks', default=use_symlinks,
                            action='store_true', dest='symlinks',
                            help='Try to use symlinks rather than copies, '
                                 'when symlinks are not the default for '
                                 'the platform.')
        parser.add_argument('--clear', default=False, action='store_true',
                            dest='clear', help='Delete the contents of the '
                                               'virtual environment '
                                               'directory if it already '
                                               'exists, before virtual '
                                               'environment creation.')
        parser.add_argument('--upgrade', default=False, action='store_true',
                            dest='upgrade', help='Upgrade the virtual '
                                                 'environment directory to '
                                                 'use this version of '
                                                 'Python, assuming Python '
                                                 'has been upgraded '
                                                 'in-place.')
        parser.add_argument('--verbose', default=False, action='store_true',
                            dest='verbose', help='Display the output '
                                                 'from the scripts which '
                                                 'install setuptools and pip.')
        options = parser.parse_args(args)
        if options.upgrade and options.clear:
            raise ValueError('you cannot supply --upgrade and --clear together.')
```

(continua na próxima página)

(continuação da página anterior)

```
        builder = ExtendedEnvBuilder(system_site_packages=options.system_site,
                                     clear=options.clear,
                                     symlinks=options.symlinks,
                                     upgrade=options.upgrade,
                                     nodist=options.nodist,
                                     nopip=options.nopip,
                                     verbose=options.verbose)
        for d in options.dirs:
            builder.create(d)

if __name__ == '__main__':
    rc = 1
    try:
        main()
        rc = 0
    except Exception as e:
        print('Error: %s' % e, file=sys.stderr)
    sys.exit(rc)
```

Esse script também está disponível para download online.

29.4 `zipapp` — Manage executable Python zip archives

Novo na versão 3.5.

Código Fonte: Lib/zipapp.py

This module provides tools to manage the creation of zip files containing Python code, which can be executed directly by the Python interpreter. The module provides both a *Interface de Linha de Comando* and a *API Python*.

29.4.1 Basic Example

The following example shows how the *Interface de Linha de Comando* can be used to create an executable archive from a directory containing Python code. When run, the archive will execute the `main` function from the module `myapp` in the archive.

```
$ python -m zipapp myapp -m "myapp:main"
$ python myapp.pyz
<output from myapp>
```

29.4.2 Interface de Linha de Comando

When called as a program from the command line, the following form is used:

```
$ python -m zipapp source [options]
```

If *source* is a directory, this will create an archive from the contents of *source*. If *source* is a file, it should be an archive, and it will be copied to the target archive (or the contents of its shebang line will be displayed if the –info option is specified).

The following options are understood:

-o <output>, **--output**=<output>
Write the output to a file named *output*. If this option is not specified, the output filename will be the same as the input *source*, with the extension `.pyz` added. If an explicit filename is given, it is used as is (so a `.pyz` extension should be included if required).

An output filename must be specified if the *source* is an archive (and in that case, *output* must not be the same as *source*).

-p <interpreter>, **--python**=<interpreter>
Add a `#!` line to the archive specifying *interpreter* as the command to run. Also, on POSIX, make the archive executable. The default is to write no `#!` line, and not make the file executable.

-m <mainfn>, **--main**=<mainfn>
Write a `__main__.py` file to the archive that executes *mainfn*. The *mainfn* argument should have the form "pkg.mod:fn", where "pkg.mod" is a package/module in the archive, and "fn" is a callable in the given module. The `__main__.py` file will execute that callable.

--main cannot be specified when copying an archive.

-c, --compress
Compress files with the deflate method, reducing the size of the output file. By default, files are stored uncompressed in the archive.

--compress has no effect when copying an archive.

Novo na versão 3.7.

--info
Display the interpreter embedded in the archive, for diagnostic purposes. In this case, any other options are ignored and SOURCE must be an archive, not a directory.

-h, --help
Print a short usage message and exit.

29.4.3 API Python

The module defines two convenience functions:

zipapp.**create_archive**(*source*, *target=None*, *interpreter=None*, *main=None*, *filter=None*, *compressed=False*)
Create an application archive from *source*. The source can be any of the following:

- The name of a directory, or a *path-like object* referring to a directory, in which case a new application archive will be created from the content of that directory.
- The name of an existing application archive file, or a *path-like object* referring to such a file, in which case the file is copied to the target (modifying it to reflect the value given for the *interpreter* argument). The file name should include the `.pyz` extension, if required.
- A file object open for reading in bytes mode. The content of the file should be an application archive, and the file object is assumed to be positioned at the start of the archive.

The *target* argument determines where the resulting archive will be written:

- If it is the name of a file, or a *path-like object*, the archive will be written to that file.
- If it is an open file object, the archive will be written to that file object, which must be open for writing in bytes mode.
- If the target is omitted (or `None`), the source must be a directory and the target will be a file with the same name as the source, with a `.pyz` extension added.

The *interpreter* argument specifies the name of the Python interpreter with which the archive will be executed. It is written as a "shebang" line at the start of the archive. On POSIX, this will be interpreted by the OS, and on Windows it will be handled by the Python launcher. Omitting the *interpreter* results in no shebang line being written. If an interpreter is specified, and the target is a filename, the executable bit of the target file will be set.

The *main* argument specifies the name of a callable which will be used as the main program for the archive. It can only be specified if the source is a directory, and the source does not already contain a `__main__.py` file. The *main* argument should take the form "pkg.module:callable" and the archive will be run by importing "pkg.module" and executing the given callable with no arguments. It is an error to omit *main* if the source is a directory and does not contain a `__main__.py` file, as otherwise the resulting archive would not be executable.

The optional *filter* argument specifies a callback function that is passed a Path object representing the path to the file being added (relative to the source directory). It should return `True` if the file is to be added.

The optional *compressed* argument determines whether files are compressed. If set to `True`, files in the archive are compressed with the deflate method; otherwise, files are stored uncompressed. This argument has no effect when copying an existing archive.

If a file object is specified for *source* or *target*, it is the caller's responsibility to close it after calling create_archive.

When copying an existing archive, file objects supplied only need `read` and `readline`, or `write` methods. When creating an archive from a directory, if the target is a file object it will be passed to the `zipfile.ZipFile` class, and must supply the methods needed by that class.

Novo na versão 3.7: Added the *filter* and *compressed* arguments.

`zipapp.`**`get_interpreter`**(*archive*)

Return the interpreter specified in the `#!` line at the start of the archive. If there is no `#!` line, return *None*. The *archive* argument can be a filename or a file-like object open for reading in bytes mode. It is assumed to be at the start of the archive.

29.4.4 Exemplos

Pack up a directory into an archive, and run it.

```
$ python -m zipapp myapp
$ python myapp.pyz
<output from myapp>
```

The same can be done using the *create_archive()* function:

```
>>> import zipapp
>>> zipapp.create_archive('myapp', 'myapp.pyz')
```

To make the application directly executable on POSIX, specify an interpreter to use.

```
$ python -m zipapp myapp -p "/usr/bin/env python"
$ ./myapp.pyz
<output from myapp>
```

To replace the shebang line on an existing archive, create a modified archive using the *create_archive()* function:

```
>>> import zipapp
>>> zipapp.create_archive('old_archive.pyz', 'new_archive.pyz', '/usr/bin/python3')
```

To update the file in place, do the replacement in memory using a `BytesIO` object, and then overwrite the source afterwards. Note that there is a risk when overwriting a file in place that an error will result in the loss of the original file.

This code does not protect against such errors, but production code should do so. Also, this method will only work if the archive fits in memory:

```
>>> import zipapp
>>> import io
>>> temp = io.BytesIO()
>>> zipapp.create_archive('myapp.pyz', temp, '/usr/bin/python2')
>>> with open('myapp.pyz', 'wb') as f:
>>>     f.write(temp.getvalue())
```

29.4.5 Especificando o interpretador

Note that if you specify an interpreter and then distribute your application archive, you need to ensure that the interpreter used is portable. The Python launcher for Windows supports most common forms of POSIX `#!` line, but there are other issues to consider:

- If you use "/usr/bin/env python" (or other forms of the "python" command, such as "/usr/bin/python"), you need to consider that your users may have either Python 2 or Python 3 as their default, and write your code to work under both versions.
- If you use an explicit version, for example "/usr/bin/env python3" your application will not work for users who do not have that version. (This may be what you want if you have not made your code Python 2 compatible).
- There is no way to say "python X.Y or later", so be careful of using an exact version like "/usr/bin/env python3.4" as you will need to change your shebang line for users of Python 3.5, for example.

Typically, you should use an "/usr/bin/env python2" or "/usr/bin/env python3", depending on whether your code is written for Python 2 or 3.

29.4.6 Creating Standalone Applications with zipapp

Using the *zipapp* module, it is possible to create self-contained Python programs, which can be distributed to end users who only need to have a suitable version of Python installed on their system. The key to doing this is to bundle all of the application's dependencies into the archive, along with the application code.

The steps to create a standalone archive are as follows:

1. Create your application in a directory as normal, so you have a `myapp` directory containing a `__main__.py` file, and any supporting application code.
2. Install all of your application's dependencies into the `myapp` directory, using pip:

   ```
   $ python -m pip install -r requirements.txt --target myapp
   ```

   (this assumes you have your project requirements in a `requirements.txt` file - if not, you can just list the dependencies manually on the pip command line).
3. Optionally, delete the `.dist-info` directories created by pip in the `myapp` directory. These hold metadata for pip to manage the packages, and as you won't be making any further use of pip they aren't required - although it won't do any harm if you leave them.
4. Empacote a aplicação usando:

   ```
   $ python -m zipapp -p "interpreter" myapp
   ```

This will produce a standalone executable, which can be run on any machine with the appropriate interpreter available. See *Especificando o interpretador* for details. It can be shipped to users as a single file.

On Unix, the `myapp.pyz` file is executable as it stands. You can rename the file to remove the `.pyz` extension if you prefer a "plain" command name. On Windows, the `myapp.pyz[w]` file is executable by virtue of the fact that the Python interpreter registers the `.pyz` and `.pyzw` file extensions when installed.

Making a Windows executable

On Windows, registration of the `.pyz` extension is optional, and furthermore, there are certain places that don't recognise registered extensions "transparently" (the simplest example is that `subprocess.run(['myapp'])` won't find your application - you need to explicitly specify the extension).

On Windows, therefore, it is often preferable to create an executable from the zipapp. This is relatively easy, although it does require a C compiler. The basic approach relies on the fact that zipfiles can have arbitrary data prepended, and Windows exe files can have arbitrary data appended. So by creating a suitable launcher and tacking the `.pyz` file onto the end of it, you end up with a single-file executable that runs your application.

A suitable launcher can be as simple as the following:

```
#define Py_LIMITED_API 1
#include "Python.h"

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

#ifdef WINDOWS
int WINAPI wWinMain(
    HINSTANCE hInstance,      /* handle to current instance */
    HINSTANCE hPrevInstance,  /* handle to previous instance */
    LPWSTR lpCmdLine,         /* pointer to command line */
    int nCmdShow              /* show state of window */
)
#else
int wmain()
#endif
{
    wchar_t **myargv = _alloca((__argc + 1) * sizeof(wchar_t*));
    myargv[0] = __wargv[0];
    memcpy(myargv + 1, __wargv, __argc * sizeof(wchar_t *));
    return Py_Main(__argc+1, myargv);
}
```

If you define the `WINDOWS` preprocessor symbol, this will generate a GUI executable, and without it, a console executable.

To compile the executable, you can either just use the standard MSVC command line tools, or you can take advantage of the fact that distutils knows how to compile Python source:

```
>>> from distutils.ccompiler import new_compiler
>>> import distutils.sysconfig
>>> import sys
>>> import os
>>> from pathlib import Path

>>> def compile(src):
>>>     src = Path(src)
>>>     cc = new_compiler()
>>>     exe = src.stem
>>>     cc.add_include_dir(distutils.sysconfig.get_python_inc())
>>>     cc.add_library_dir(os.path.join(sys.base_exec_prefix, 'libs'))
```

(continua na próxima página)

(continuação da página anterior)

```
>>>     # First the CLI executable
>>>     objs = cc.compile([str(src)])
>>>     cc.link_executable(objs, exe)
>>>     # Now the GUI executable
>>>     cc.define_macro('WINDOWS')
>>>     objs = cc.compile([str(src)])
>>>     cc.link_executable(objs, exe + 'w')

>>> if __name__ == "__main__":
>>>     compile("zastub.c")
```

The resulting launcher uses the "Limited ABI", so it will run unchanged with any version of Python 3.x. All it needs is for Python (`python3.dll`) to be on the user's `PATH`.

For a fully standalone distribution, you can distribute the launcher with your application appended, bundled with the Python "embedded" distribution. This will run on any PC with the appropriate architecture (32 bit or 64 bit).

Caveats

There are some limitations to the process of bundling your application into a single file. In most, if not all, cases they can be addressed without needing major changes to your application.

1. If your application depends on a package that includes a C extension, that package cannot be run from a zip file (this is an OS limitation, as executable code must be present in the filesystem for the OS loader to load it). In this case, you can exclude that dependency from the zipfile, and either require your users to have it installed, or ship it alongside your zipfile and add code to your `__main__.py` to include the directory containing the unzipped module in `sys.path`. In this case, you will need to make sure to ship appropriate binaries for your target architecture(s) (and potentially pick the correct version to add to `sys.path` at runtime, based on the user's machine).
2. If you are shipping a Windows executable as described above, you either need to ensure that your users have `python3.dll` on their PATH (which is not the default behaviour of the installer) or you should bundle your application with the embedded distribution.
3. The suggested launcher above uses the Python embedding API. This means that in your application, `sys.executable` will be your application, and *not* a conventional Python interpreter. Your code and its dependencies need to be prepared for this possibility. For example, if your application uses the *multiprocessing* module, it will need to call *multiprocessing.set_executable()* to let the module know where to find the standard Python interpreter.

29.4.7 The Python Zip Application Archive Format

Python has been able to execute zip files which contain a `__main__.py` file since version 2.6. In order to be executed by Python, an application archive simply has to be a standard zip file containing a `__main__.py` file which will be run as the entry point for the application. As usual for any Python script, the parent of the script (in this case the zip file) will be placed on *sys.path* and thus further modules can be imported from the zip file.

The zip file format allows arbitrary data to be prepended to a zip file. The zip application format uses this ability to prepend a standard POSIX "shebang" line to the file (`#!/path/to/interpreter`).

Formally, the Python zip application format is therefore:

1. An optional shebang line, containing the characters `b'#!'` followed by an interpreter name, and then a newline (`b'\n'`) character. The interpreter name can be anything acceptable to the OS "shebang" processing, or the Python launcher on Windows. The interpreter should be encoded in UTF-8 on Windows, and in *sys.getfilesystemencoding()* on POSIX.

2. Standard zipfile data, as generated by the *zipfile* module. The zipfile content *must* include a file called `__main__.py` (which must be in the "root" of the zipfile - i.e., it cannot be in a subdirectory). The zipfile data can be compressed or uncompressed.

If an application archive has a shebang line, it may have the executable bit set on POSIX systems, to allow it to be executed directly.

There is no requirement that the tools in this module are used to create application archives - the module is a convenience, but archives in the above format created by any means are acceptable to Python.

CAPÍTULO 30

Serviços de Tempo de Execução Python

Os módulos descritos neste capítulo oferecem uma ampla gama de serviços relacionados ao interpretador Python e sua interação com o ambiente. Aqui está uma visão geral:

30.1 `sys` — Parâmetros e funções específicas do sistema

Este módulo fornece acesso a algumas variáveis usadas ou mantidas pelo interpretador e a funções que interagem fortemente com o interpretador. Sempre disponível.

`sys.`**`abiflags`**
Em sistemas POSIX onde Python foi construído com o script `configure` padrão, ele contém os sinalizadores ABI conforme especificado por **PEP 3149**.

Novo na versão 3.2.

`sys.`**`argv`**
A lista de argumentos de linha de comando passados para um script Python. `argv[0]` é o nome do script (depende do sistema operacional se este é um nome de caminho completo ou não). Se o comando foi executado usando a opção de linha de comando `-c` para o interpretador, `argv[0]` é definido como a string `'-c'`. Se nenhum nome de script foi passado para o interpretador Python, `argv[0]` é a string vazia.

Para percorrer a entrada padrão ou a lista de arquivos fornecida na linha de comando, consulte o módulo *`fileinput`*.

Nota: No Unix, os argumentos da linha de comando são passados por bytes do sistema operacional. O Python os decodifica com a codificação do sistema de arquivos e o tratador de erros "surrogateescape". Quando você precisar de bytes originais, você pode obtê-los por `[os.fsencode(arg) for arg in sys.argv]`.

`sys.`**`base_exec_prefix`**
Definido durante a inicialização do Python, antes de `site.py` ser executado, para o mesmo valor que *`exec_prefix`*. Se não estiver executando em um *ambiente virtual*, os valores permanecerão os mesmos; se

site.py descobrir que um ambiente virtual está em uso, os valores de *prefix* e *exec_prefix* serão alterados para apontar para o ambiente virtual, enquanto *base_prefix* e *base_exec_prefix* permanecerá apontando para a instalação base do Python (aquela a partir da qual o ambiente virtual foi criado).

Novo na versão 3.3.

sys.**base_prefix**
Definido durante a inicialização do Python, antes de site.py ser executado, para o mesmo valor que *prefix*. Se não estiver executando em um *ambiente virtual*, os valores permanecerão os mesmos; se site.py descobrir que um ambiente virtual está em uso, os valores de *prefix* e *exec_prefix* serão alterados para apontar para o ambiente virtual, enquanto *base_prefix* e *base_exec_prefix* permanecerá apontando para a instalação base do Python (aquela a partir da qual o ambiente virtual foi criado).

Novo na versão 3.3.

sys.**byteorder**
Um indicador da ordem nativa de bytes. Isso terá o valor 'big' em plataformas big endian (byte mais significativo primeiro) e 'little' em plataformas little endian (byte menos significativo primeiro).

sys.**builtin_module_names**
A tuple of strings giving the names of all modules that are compiled into this Python interpreter. (This information is not available in any other way — modules.keys() only lists the imported modules.)

sys.**call_tracing**(*func*, *args*)
Chama func(*args), enquanto o rastreamento está habilitado. O estado de rastreamento é salvo e restaurado posteriormente. Isso deve ser chamado de um depurador de um ponto de verificação, para depurar recursivamente algum outro código.

sys.**copyright**
Uma string contendo os direitos autorais pertencentes ao interpretador Python.

sys.**_clear_type_cache**()
Limpa o cache de tipo interno. O cache de tipo é usado para acelerar pesquisas de atributos e métodos. Use a função *apenas* para descartar referências desnecessárias durante a depuração de vazamento de referência.

Esta função deve ser usada apenas para fins internos e especializados.

sys.**_current_frames**()
Retorna um dicionário mapeando o identificador de cada encadeamento (*thread*) para o quadro de pilha mais alto atualmente ativo nesse encadeamento no momento em que a função é chamada. Observe que as funções no módulo *traceback* podem construir a pilha de chamadas dado tal quadro.

Isso é mais útil para depurar impasses: esta função não requer a cooperação dos encadeamentos em impasse e as pilhas de chamadas de tais encadeamentos são congeladas enquanto permanecerem em impasse (*deadlock*). O quadro retornado para um encadeamento sem impasse pode não ter nenhuma relação com a atividade atual desse encadeamento no momento em que o código de chamada examina o quadro.

Esta função deve ser usada apenas para fins internos e especializados.

sys.**breakpointhook**()
Esta função de gancho é chamada pela função embutida *breakpoint()*. Por padrão, ela leva você ao depurador *pdb*, mas pode ser configurado para qualquer outra função para que você possa escolher qual depurador será usado.

A assinatura dessa função depende do que ela chama. Por exemplo, a ligação padrão (por exemplo, pdb.set_trace()) não espera nenhum argumento, mas você pode vinculá-la a uma função que espera argumentos adicionais (posicionais e/ou nomeados). A função embutida breakpoint() passa seus *args e **kws diretamente. O que quer que breakpointhooks() retorne é retornado de breakpoint().

A implementação padrão primeiro consulta a variável de ambiente PYTHONBREAKPOINT. Se for definido como "0", então esta função retorna imediatamente; ou seja, é um no-op. Se a variável de ambiente não for definida, ou for definida como uma string vazia, pdb.set_trace() será chamado. Caso contrário, essa variável deve

nomear uma função a ser executada, usando a nomenclatura de importação pontilhada do Python, por exemplo `package.subpackage.module.function`. Neste caso, `package.subpackage.module` seria importado e o módulo resultante deve ter um chamável chamado `function()`. Isso é executado, passando `*args` e `**kws`, e qualquer que seja o retorno de `function()`, `sys.breakpointhook()` retorna para a função embutida *`breakpoint()`*.

Observe que se algo der errado ao importar o chamável nomeado por `PYTHONBREAKPOINT`, uma *`RuntimeWarning`* é relatado e o ponto de interrupção é ignorado.

Observe também que se `sys.breakpointhook()` for sobrescrito programaticamente, `PYTHONBREAKPOINT` *não* será consultado.

Novo na versão 3.7.

`sys._debugmallocstats()`
Imprima informações de baixo nível para stderr sobre o estado do alocador de memória do CPython.

If Python is configured –with-pydebug, it also performs some expensive internal consistency checks.

Novo na versão 3.3.

CPython implementation detail: Esta função é específica para CPython. O formato de saída exato não é definido aqui e pode mudar.

`sys.dllhandle`
Número inteiro que especifica o identificador da DLL do Python.

Disponibilidade: Windows.

`sys.displayhook(value)`
Se *value* não for `None`, esta função imprime `repr(value)` em `sys.stdout`, e salva *value* em `builtins._`. Se `repr(value)` não for codificável para `sys.stdout.encoding` com o tratador de erros `sys.stdout.errors` (que provavelmente é `'strict'`), codifica-o para `sys.stdout.encoding` com tratador de erros `'backslashreplace'`.

`sys.displayhook` é chamado no resultado da avaliação de uma *expressão* inserida em uma sessão interativa do Python. A exibição desses valores pode ser personalizada atribuindo outra função de um argumento a `sys.displayhook`.

Pseudocódigo:

```
def displayhook(value):
    if value is None:
        return
    # Set '_' to None to avoid recursion
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
        bytes = text.encode(sys.stdout.encoding, 'backslashreplace')
        if hasattr(sys.stdout, 'buffer'):
            sys.stdout.buffer.write(bytes)
        else:
            text = bytes.decode(sys.stdout.encoding, 'strict')
            sys.stdout.write(text)
    sys.stdout.write("\n")
    builtins._ = value
```

Alterado na versão 3.2: Usa o tratador de erros `'backslashreplace'` ao ser levantada *`UnicodeEncodeError`*.

sys.**dont_write_bytecode**
Se isso for true, o Python não tentará escrever arquivos `.pyc` na importação de módulos fonte. Este valor é inicialmente definido como `True` ou `False` dependendo da opção de linha de comando `-B` e da variável de ambiente `PYTHONDONTWRITEBYTECODE`, mas você mesmo pode configurá-lo para controlar geração de arquivo bytecode.

sys.**excepthook**(*type*, *value*, *traceback*)
Esta função imprime um determinado traceback (situação da pilha de execução) e exceção para `sys.stderr`.

Quando uma exceção é lançada e não capturada, o interpretador chama `sys.excepthook` com três argumentos, a classe de exceção, a instância de exceção e um objeto traceback. Em uma sessão interativa, isso acontece logo antes de o controle retornar ao prompt; em um programa Python, isso acontece pouco antes de o programa ser encerrado. A manipulação de tais exceções de nível superior pode ser personalizada atribuindo outra função de três argumentos a `sys.excepthook`.

sys.**__breakpointhook__**
sys.**__displayhook__**
sys.**__excepthook__**
These objects contain the original values of `breakpointhook`, `displayhook`, and `excepthook` at the start of the program. They are saved so that `breakpointhook`, `displayhook` and `excepthook` can be restored in case they happen to get replaced with broken or alternative objects.

Novo na versão 3.7: __breakpointhook__

sys.**exc_info**()
This function returns a tuple of three values that give information about the exception that is currently being handled. The information returned is specific both to the current thread and to the current stack frame. If the current stack frame is not handling an exception, the information is taken from the calling stack frame, or its caller, and so on until a stack frame is found that is handling an exception. Here, "handling an exception" is defined as "executing an except clause." For any stack frame, only information about the exception being currently handled is accessible.

If no exception is being handled anywhere on the stack, a tuple containing three `None` values is returned. Otherwise, the values returned are `(type, value, traceback)`. Their meaning is: *type* gets the type of the exception being handled (a subclass of *`BaseException`*); *value* gets the exception instance (an instance of the exception type); *traceback* gets a traceback object (see the Reference Manual) which encapsulates the call stack at the point where the exception originally occurred.

sys.**exec_prefix**
Uma string que fornece o prefixo do diretório específico do site onde os arquivos Python dependentes da plataforma são instalados; por padrão, também é `'/usr/local'`. Isso pode ser definido em tempo de compilação com o argumento `--exec-prefix` para o script **configure**. Especificamente, todos os arquivos de configuração (por exemplo, o arquivo de cabeçalho `pyconfig.h`) são instalados no diretório *exec_prefix*`/lib/python`*X.Y*`/config`, e os módulos da biblioteca compartilhada são instalados em *exec_prefix*`/lib/python`*X.Y*`/lib-dynload`, onde *X.Y* é o número da versão do Python, por exemplo `3.2`.

Nota: Se um *ambiente virtual* estiver em vigor, este valor será alterado em `site.py` para apontar para o ambiente virtual. O valor para a instalação do Python ainda estará disponível, via *`base_exec_prefix`*.

---

sys.**executable**
Uma string que fornece o caminho absoluto do binário executável para o interpretador Python, em sistemas onde isso faz sentido. Se o Python não conseguir recuperar o caminho real para seu executável, *`sys.executable`* será uma string vazia ou `None`.

sys.**exit**([*arg*])
Exit from Python. This is implemented by raising the *`SystemExit`* exception, so cleanup actions specified by finally clauses of `try` statements are honored, and it is possible to intercept the exit attempt at an outer level.

O argumento opcional *arg* pode ser um número inteiro dando o status de saída (o padrão é zero) ou outro tipo de objeto. Se for um número inteiro, zero é considerado "terminação bem-sucedida" e qualquer valor diferente de zero é considerado "terminação anormal" por shells e similares. A maioria dos sistemas exige que ele esteja no intervalo de 0 a 127 e, caso contrário, produz resultados indefinidos. Alguns sistemas têm uma convenção para atribuir significados específicos a códigos de saída específicos, mas estes são geralmente subdesenvolvidos; Programas Unix geralmente usam 2 para erros de sintaxe de linha de comando e 1 para todos os outros tipos de erros. Se outro tipo de objeto for passado, `None` é equivalente a passar zero, e qualquer outro objeto é impresso em *stderr* e resulta em um código de saída de 1. Em particular, `sys.exit( "alguma mensagem de erro")` é uma maneira rápida de sair de um programa quando ocorre um erro.

Since *exit()* ultimately "only" raises an exception, it will only exit the process when called from the main thread, and the exception is not intercepted.

Alterado na versão 3.6: Se ocorrer um erro na limpeza após o interpretador Python ter capturado *SystemExit* (como um erro ao liberar dados em buffer nos fluxos padrão), o status de saída é alterado para 120.

`sys.`**`flags`**

A tupla nomeada *flags* expõe o status dos sinalizadores de linha de comando. Os atributos são somente leitura.

atributo	sinalizador
`debug`	`-d`
inspect	`-i`
`interactive`	`-i`
`isolated`	`-I`
`optimize`	`-O` or `-OO`
dont_write_bytecode	`-B`
`no_user_site`	`-s`
`no_site`	`-S`
`ignore_environment`	`-E`
`verbose`	`-v`
`bytes_warning`	`-b`
`quiet`	`-q`
`hash_randomization`	`-R`
`dev_mode`	`-X dev`
`utf8_mode`	`-X utf8`
`int_max_str_digits`	`-X int_max_str_digits` (*integer string conversion length limitation*)

Alterado na versão 3.2: Adicionado o atributo `quiet` para o novo sinalizador `-q`.

Novo na versão 3.2.3: O atributo `hash_randomization`.

Alterado na versão 3.3: Removido o atributo obsoleto `division_warning`.

Alterado na versão 3.4: Adicionado o atributo `isolated` para o sinalizador:option:-*I* `isolated`.

Alterado na versão 3.7: Added `dev_mode` attribute for the new `-X dev` flag and `utf8_mode` attribute for the new `-X utf8` flag.

Alterado na versão 3.7.14: Added the `int_max_str_digits` attribute.

`sys.`**`float_info`**

Uma tupla nomeada contendo informações sobre o tipo float, ponto flutuante. Ele contém informações de baixo nível sobre a precisão e a representação interna. Os valores correspondem às várias constantes de ponto flutuante definidas no arquivo de cabeçalho padrão `float.h` para a linguagem de programação 'C'; consulte a seção 5.2.4.2.2 do padrão ISO/IEC C de 1999 [C99], 'Características dos tipos flutuantes', para obter detalhes.

atributo	macro em float.h	explicação
`epsilon`	DBL_EPSILON	diferença entre 1,0 e o menor valor maior que 1,0 que pode ser representado como ponto flutuante
`dig`	DBL_DIG	número máximo de dígitos decimais que podem ser fielmente representados em um ponto flutuante; Veja abaixo
`mant_dig`	DBL_MANT_DIG	precisão do ponto flutuante: o número de dígitos de base `radix` no significando de um ponto flutuante
max	DBL_MAX	ponto flutuante finito positivo máximo representável
`max_exp`	DBL_MAX_EXP	inteiro máximo *e* de tal modo que“radix**(e-1)“ é um ponto flutuante finito representável
`max_10_exp`	DBL_MAX_10_EXP	inteiro máximo *e* de tal modo que“10**e“ é um intervalo de pontos flutuantes finitos representáveis
min	DBL_MIN	ponto flutuante *normalizado* positivo mínimo representável
`min_exp`	DBL_MIN_EXP	inteiro mínimo *e* de tal modo que `radix**(e-1)` é um ponto flutuante normalizado
`min_10_exp`	DBL_MIN_10_EXP	inteiro mínimo *e* de tal modo que `10**e` é um ponto flutuante normalizado
`radix`	FLT_RADIX	raiz da representação do expoente
`rounds`	FLT_ROUNDS	integer constant representing the rounding mode used for arithmetic operations. This reflects the value of the system FLT_ROUNDS macro at interpreter startup time. See section 5.2.4.2.2 of the C99 standard for an explanation of the possible values and their meanings.

O atributo `sys.float_info.dig` precisa de mais explicações. Se `s` for qualquer string representando um número decimal com no máximo `sys.float_info.dig` dígitos significativos, então converter `s` para ponto flutuante e vice-versa recuperará uma string representando o mesmo decimal valor:

```
>>> import sys
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'    # decimal string with 15 significant digits
>>> format(float(s), '.15g')  # convert to float and back -> same value
'3.14159265358979'
```

Mas para strings com mais de `sys.float_info.dig` dígitos significativos, isso nem sempre é verdade:

```
>>> s = '9876543211234567'    # 16 significant digits is too many!
>>> format(float(s), '.16g')  # conversion changes value
'9876543211234568'
```

`sys.`**`float_repr_style`**

Uma string indicando como a função *repr()* se comporta para floats. Se a string tem valor `'short'` então para um ponto flutuante finito `x`, `repr(x)` visa produzir uma string curta com a propriedade que `float(repr(x)) == x`. Este é o comportamento usual no Python 3.1 e posterior. Caso contrário, `float_repr_style` tem valor `'legacy'` e `repr(x)` se comporta da mesma forma que nas versões do Python anteriores a 3.1.

Novo na versão 3.1.

`sys.`**`getallocatedblocks`**`()`

Retorna o número de blocos de memória atualmente alocados pelo interpretador, independentemente de seu tamanho. Esta função é útil principalmente para rastrear e depurar vazamentos de memória. Devido aos caches internos do interpretador, o resultado pode variar de chamada para chamada; você pode ter que chamar *_clear_type_cache()* e *gc.collect()* para obter resultados mais previsíveis.

Se uma construção ou implementação do Python não puder computar razoavelmente essas informações, *getallocatedblocks()* poderá retornar 0 em seu lugar.

Novo na versão 3.4.

sys.**getandroidapilevel**()
Retorna a versão da API de tempo de compilação do Android como um número inteiro.

Disponibilidade: Android.

Novo na versão 3.7.

sys.**getcheckinterval**()
Return the interpreter's "check interval"; see *setcheckinterval()*.

Obsoleto desde a versão 3.2: Use *getswitchinterval()* instead.

sys.**getdefaultencoding**()
Retorna o nome da codificação de string padrão atual usada pela implementação Unicode.

sys.**getdlopenflags**()
Retorna o valor atual dos sinalizadores que são usados para chamadas `dlopen()`. Nomes simbólicos para os valores dos sinalizadores podem ser encontrados no módulo *os* (constantes `RTLD_xxx`, por exemplo *os.RTLD_LAZY*).

Disponibilidade: Unix.

sys.**getfilesystemencoding**()
Return the name of the encoding used to convert between Unicode filenames and bytes filenames. For best compatibility, str should be used for filenames in all cases, although representing filenames as bytes is also supported. Functions accepting or returning filenames should support either str or bytes and internally convert to the system's preferred representation.

This encoding is always ASCII-compatible.

os.fsencode() e *os.fsdecode()* devem ser usados para garantir que a codificação correta e o modo de erros sejam usados.

- In the UTF-8 mode, the encoding is `utf-8` on any platform.
- On Mac OS X, the encoding is `'utf-8'`.
- On Unix, the encoding is the locale encoding.
- On Windows, the encoding may be `'utf-8'` or `'mbcs'`, depending on user configuration.

Alterado na versão 3.2: O resultado de *getfilesystemencoding()* não pode mais ser `None`.

Alterado na versão 3.6: O Windows não tem mais garantia de retornar `'mbcs'`. Veja **PEP 529** e *_enablelegacywindowsfsencoding()* para mais informações.

Alterado na versão 3.7: Return 'utf-8' in the UTF-8 mode.

sys.**getfilesystemencodeerrors**()
Return the name of the error mode used to convert between Unicode filenames and bytes filenames. The encoding name is returned from *getfilesystemencoding()*.

os.fsencode() e *os.fsdecode()* devem ser usados para garantir que a codificação correta e o modo de erros sejam usados.

Novo na versão 3.6.

sys.**get_int_max_str_digits**()
Returns the current value for the *integer string conversion length limitation*. See also *set_int_max_str_digits()*.

Novo na versão 3.7.14.

sys.**getrefcount**(*object*)
Retorna a contagem de referências do *object*. A contagem retornada é geralmente um valor maior do que o esperado, porque inclui a referência (temporária) como um argumento para *getrefcount()*.

sys.**getrecursionlimit**()
Retorna o valor atual do limite de recursão, a profundidade máxima da pilha do interpretador Python. Esse limite evita que a recursão infinita cause um estouro da pilha C e falhe o Python. Pode ser definido por *setrecursionlimit()*.

sys.**getsizeof**(*object*[, *default*])
Retorna o tamanho de um objeto em bytes. O objeto pode ser qualquer tipo de objeto. Todos os objetos embutidos retornarão resultados corretos, mas isso não precisa ser verdadeiro para extensões de terceiros, pois é específico da implementação.

Apenas o consumo de memória diretamente atribuído ao objeto é contabilizado, não o consumo de memória dos objetos a que ele se refere.

Se fornecido, *default* será retornado se o objeto não fornecer meios para recuperar o tamanho. Caso contrário, uma exceção *TypeError* será levantada.

getsizeof() chama o método __sizeof__ do objeto e adiciona uma sobrecarga adicional do coletor de lixo se o objeto for gerenciado pelo coletor de lixo.

Consulte receita de tamanho recursivo para obter um exemplo de uso de *getsizeof()* recursivamente para encontrar o tamanho dos contêineres e todo o seu conteúdo.

sys.**getswitchinterval**()
Retorna o "intervalo de troca de thread" do interpretador; veja *setswitchinterval()*.

Novo na versão 3.2.

sys.**_getframe**([*depth*])
Retorna um objeto quadro da pilha de chamadas. Se o inteiro opcional *depth* for fornecido, retorna o objeto de quadro que muitos chamam abaixo do topo da pilha. Se for mais profundo do que a pilha de chamadas, *ValueError* é levantada. O padrão para *depth* é zero, retornando o quadro no topo da pilha de chamadas.

CPython implementation detail: Esta função deve ser usada apenas para fins internos e especializados. Não é garantido que exista em todas as implementações do Python.

sys.**getprofile**()
Obtém a função do criador de perfil conforme definido por *setprofile()*.

sys.**gettrace**()
Obtém a função trace conforme definido por *settrace()*.

CPython implementation detail: A função *gettrace()* destina-se apenas à implementação de depuradores, criadores de perfil, ferramentas de cobertura e similares. Seu comportamento faz parte da plataforma de implementação, e não da definição da linguagem e, portanto, pode não estar disponível em todas as implementações do Python.

sys.**getwindowsversion**()
Retorna uma tupla nomeada que descreve a versão do Windows em execução no momento. Os elementos nomeados são *major*, *minor*, *build*, *platform*, *service_pack*, *service_pack_minor*, *service_pack_major*, *suite_mask*, *product_type* e *platform_version*. *service_pack* contém uma string, *platform_version* uma tupla de 3 elementos e todos os outros valores são inteiros. Os componentes também podem ser acessados pelo nome, então `sys.getwindowsversion()[0]` é equivalente a `sys.getwindowsversion().major`. Para compatibilidade com versões anteriores, apenas os primeiros 5 elementos são recuperáveis por indexação.

platform será `2 (VER_PLATFORM_WIN32_NT)`.

product_type pode ser um dos seguintes valores:

Constante	Significado
1 (VER_NT_WORKSTATION)	O sistema é uma estação de trabalho
2 (VER_NT_DOMAIN_CONTROLLER)	O sistema é um controlador de domínio.
3 (VER_NT_SERVER)	O sistema é um servidor, mas não um controlador de domínio.

Esta função atua como um envólucro em volta da função Win32 `GetVersionEx()`; consulte a documentação da Microsoft em `OSVERSIONINFOEX()` para obter mais informações sobre esses campos.

platform_version returns the accurate major version, minor version and build number of the current operating system, rather than the version that is being emulated for the process. It is intended for use in logging rather than for feature detection.

Disponibilidade: Windows.

Alterado na versão 3.2: Alterado para uma tupla nomeada e adicionado *service_pack_minor*, *service_pack_major*, *suite_mask* e *product_type*.

Alterado na versão 3.6: Adicionado *platform_version*

sys.**get_asyncgen_hooks**()
Returns an *asyncgen_hooks* object, which is similar to a *namedtuple* of the form *(firstiter, finalizer)*, where *firstiter* and *finalizer* are expected to be either `None` or functions which take an *asynchronous generator iterator* as an argument, and are used to schedule finalization of an asynchronous generator by an event loop.

Novo na versão 3.6: Veja **PEP 525** para mais detalhes.

Nota: Esta função foi adicionada provisoriamente (veja **PEP 411** para detalhes.)

---

sys.**get_coroutine_origin_tracking_depth**()
Obtém a profundidade de rastreamento da origem da corrotina atual, conforme definido por *set_coroutine_origin_tracking_depth()*.

Novo na versão 3.7.

Nota: Esta função foi adicionada provisoriamente (veja **PEP 411** para detalhes.) Use-a apenas para propósitos de depuração.

---

sys.**get_coroutine_wrapper**()
Returns `None`, or a wrapper set by *set_coroutine_wrapper()*.

Novo na versão 3.5: See **PEP 492** for more details.

Nota: Esta função foi adicionada provisoriamente (veja **PEP 411** para detalhes.) Use-a apenas para propósitos de depuração.

Obsoleto desde a versão 3.7: The coroutine wrapper functionality has been deprecated, and will be removed in 3.8. See bpo-32591 for details.

sys.**hash_info**
Uma tupla nomeada fornecendo parâmetros da implementação de hash numérico. Para mais detalhes sobre hashing de tipos numéricos, veja *Hashing de tipos numéricos*.

atributo	explicação
`width`	largura em bits usada para fazer hash de valores
`modulus`	módulo primo P usado para esquema de hash numérico
`inf`	valor de hash retornado para um infinito positivo
`nan`	hash value returned for a nan
`imag`	multiplicador usado para a parte imaginária de um número complexo
`algorithm`	nome do algoritmo para hash de str, bytes e memoryview
`hash_bits`	tamanho da saída interna do algoritmo de hash
`seed_bits`	tamanho da chave semente do algoritmo hash

Novo na versão 3.2.

Alterado na versão 3.4: Adicionado *algorithm*, *hash_bits* e *seed_bits*

`sys.`**`hexversion`**
O número da versão codificado como um único inteiro. Isso é garantido para aumentar com cada versão, incluindo suporte adequado para lançamentos de não produção. Por exemplo, para testar se o interpretador Python é pelo menos a versão 1.5.2, use:

```
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

Isso é chamado `hexversion` já que só parece realmente significativo quando visto como o resultado de passá-lo para a função embutida *`hex()`*. A tupla nomeada *`sys.version_info`* pode ser usada para uma codificação mais amigável das mesmas informações.

Mais detalhes sobre `hexversion` podem ser encontrados em apiabiversion.

`sys.`**`implementation`**
Um objeto que contém informações sobre a implementação do interpretador Python em execução no momento. Os atributos a seguir devem existir em todas as implementações do Python.

name é o identificador da implementação, por exemplo `'cpython'`. A string real é definida pela implementação do Python, mas é garantido que seja minúscula.

version is a named tuple, in the same format as *`sys.version_info`*. It represents the version of the Python *implementation*. This has a distinct meaning from the specific version of the Python *language* to which the currently running interpreter conforms, which `sys.version_info` represents. For example, for PyPy 1.8 `sys.implementation.version` might be `sys.version_info(1, 8, 0, 'final', 0)`, whereas `sys.version_info` would be `sys.version_info(2, 7, 2, 'final', 0)`. For CPython they are the same value, since it is the reference implementation.

hexversion is the implementation version in hexadecimal format, like *`sys.hexversion`*.

cache_tag is the tag used by the import machinery in the filenames of cached modules. By convention, it would be a composite of the implementation's name and version, like `'cpython-33'`. However, a Python implementation may use some other value if appropriate. If `cache_tag` is set to `None`, it indicates that module caching should be disabled.

`sys.implementation` may contain additional attributes specific to the Python implementation. These non-standard attributes must start with an underscore, and are not described here. Regardless of its contents, *`sys.implementation`* will not change during a run of the interpreter, nor between implementation versions. (It may change between Python language versions, however.) See **PEP 421** for more information.

Novo na versão 3.3.

Nota: The addition of new required attributes must go through the normal PEP process. See **PEP 421** for more information.

---

sys.**int_info**

A *named tuple* that holds information about Python's internal representation of integers. The attributes are read only.

Atributo	Explanação
`bits_per_digit`	number of bits held in each digit. Python integers are stored internally in base `2**int_info.bits_per_digit`
`sizeof_digit`	size in bytes of the C type used to represent a digit
`default_max_str_digits`	default value for *`sys.get_int_max_str_digits()`* when it is not otherwise explicitly configured.
`str_digits_check_threshold`	minimum non-zero value for *`sys.set_int_max_str_digits()`*, `PYTHONINTMAXSTRDIGITS`, or `-X int_max_str_digits`.

Novo na versão 3.1.

Alterado na versão 3.7.14: Added `default_max_str_digits` and `str_digits_check_threshold`.

sys.**__interactivehook__**

When this attribute exists, its value is automatically called (with no arguments) when the interpreter is launched in interactive mode. This is done after the `PYTHONSTARTUP` file is read, so that you can set this hook there. The *`site`* module *sets this*.

Novo na versão 3.4.

sys.**intern**(*string*)

Enter *string* in the table of "interned" strings and return the interned string – which is *string* itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup – if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys.

Interned strings are not immortal; you must keep a reference to the return value of *`intern()`* around to benefit from it.

sys.**is_finalizing**()

Return *`True`* if the Python interpreter is *shutting down*, *`False`* otherwise.

Novo na versão 3.5.

sys.**last_type**
sys.**last_value**
sys.**last_traceback**

These three variables are not always defined; they are set when an exception is not handled and the interpreter prints an error message and a stack traceback. Their intended use is to allow an interactive user to import a debugger module and engage in post-mortem debugging without having to re-execute the command that caused the error. (Typical use is `import pdb; pdb.pm()` to enter the post-mortem debugger; see *`pdb`* module for more information.)

The meaning of the variables is the same as that of the return values from *`exc_info()`* above.

sys.**maxsize**
An integer giving the maximum value a variable of type `Py_ssize_t` can take. It's usually `2**31 - 1` on a 32-bit platform and `2**63 - 1` on a 64-bit platform.

sys.**maxunicode**
An integer giving the value of the largest Unicode code point, i.e. `1114111` (`0x10FFFF` in hexadecimal).

Alterado na versão 3.3: Before **PEP 393**, `sys.maxunicode` used to be either `0xFFFF` or `0x10FFFF`, depending on the configuration option that specified whether Unicode characters were stored as UCS-2 or UCS-4.

sys.**meta_path**
A list of *meta path finder* objects that have their *find_spec()* methods called to see if one of the objects can find the module to be imported. The *find_spec()* method is called with at least the absolute name of the module being imported. If the module to be imported is contained in a package, then the parent package's `__path__` attribute is passed in as a second argument. The method returns a *module spec*, or `None` if the module cannot be found.

Ver também:

importlib.abc.MetaPathFinder The abstract base class defining the interface of finder objects on *meta_path*.

importlib.machinery.ModuleSpec The concrete class which *find_spec()* should return instances of.

Alterado na versão 3.4: *Module specs* were introduced in Python 3.4, by **PEP 451**. Earlier versions of Python looked for a method called *find_module()*. This is still called as a fallback if a *meta_path* entry doesn't have a *find_spec()* method.

sys.**modules**
This is a dictionary that maps module names to modules which have already been loaded. This can be manipulated to force reloading of modules and other tricks. However, replacing the dictionary will not necessarily work as expected and deleting essential items from the dictionary may cause Python to fail.

sys.**path**
A list of strings that specifies the search path for modules. Initialized from the environment variable `PYTHONPATH`, plus an installation-dependent default.

As initialized upon program startup, the first item of this list, `path[0]`, is the directory containing the script that was used to invoke the Python interpreter. If the script directory is not available (e.g. if the interpreter is invoked interactively or if the script is read from standard input), `path[0]` is the empty string, which directs Python to search modules in the current directory first. Notice that the script directory is inserted *before* the entries inserted as a result of `PYTHONPATH`.

A program is free to modify this list for its own purposes. Only strings and bytes should be added to *sys.path*; all other data types are ignored during import.

Ver também:

Module *site* This describes how to use .pth files to extend *sys.path*.

sys.**path_hooks**
A list of callables that take a path argument to try to create a *finder* for the path. If a finder can be created, it is to be returned by the callable, else raise *ImportError*.

Originally specified in **PEP 302**.

sys.**path_importer_cache**
A dictionary acting as a cache for *finder* objects. The keys are paths that have been passed to *sys.path_hooks* and the values are the finders that are found. If a path is a valid file system path but no finder is found on *sys.path_hooks* then `None` is stored.

Originally specified in **PEP 302**.

Alterado na versão 3.3: `None` is stored instead of *imp.NullImporter* when no finder is found.

`sys.`**platform**
This string contains a platform identifier that can be used to append platform-specific components to *sys.path*, for instance.

For Unix systems, except on Linux, this is the lowercased OS name as returned by `uname -s` with the first part of the version as returned by `uname -r` appended, e.g. `'sunos5'` or `'freebsd8'`, *at the time when Python was built*. Unless you want to test for a specific system version, it is therefore recommended to use the following idiom:

```
if sys.platform.startswith('freebsd'):
    # FreeBSD-specific code here...
elif sys.platform.startswith('linux'):
    # Linux-specific code here...
```

For other systems, the values are:

System	`platform` value
Linux	`'linux'`
Windows	`'win32'`
Windows/Cygwin	`'cygwin'`
Mac OS X	`'darwin'`

Alterado na versão 3.3: On Linux, *sys.platform* doesn't contain the major version anymore. It is always `'linux'`, instead of `'linux2'` or `'linux3'`. Since older Python versions include the version number, it is recommended to always use the `startswith` idiom presented above.

Ver também:

os.name has a coarser granularity. *os.uname()* gives system-dependent version information.

O módulo *platform* fornece verificações detalhadas sobre a identificação do sistema.

`sys.`**prefix**
A string giving the site-specific directory prefix where the platform independent Python files are installed; by default, this is the string `'/usr/local'`. This can be set at build time with the `--prefix` argument to the **configure** script. The main collection of Python library modules is installed in the directory *prefix*`/lib/python`*X.Y* while the platform independent header files (all except `pyconfig.h`) are stored in *prefix*`/include/python`*X.Y*, where *X.Y* is the version number of Python, for example `3.2`.

Nota: If a *virtual environment* is in effect, this value will be changed in `site.py` to point to the virtual environment. The value for the Python installation will still be available, via *base_prefix*.

`sys.`**ps1**
`sys.`**ps2**
Strings specifying the primary and secondary prompt of the interpreter. These are only defined if the interpreter is in interactive mode. Their initial values in this case are `'>>> '` and `'... '`. If a non-string object is assigned to either variable, its *str()* is re-evaluated each time the interpreter prepares to read a new interactive command; this can be used to implement a dynamic prompt.

`sys.`**setcheckinterval**(*interval*)
Set the interpreter's "check interval". This integer value determines how often the interpreter checks for periodic things such as thread switches and signal handlers. The default is `100`, meaning the check is performed every

100 Python virtual instructions. Setting it to a larger value may increase performance for programs using threads. Setting it to a value <= 0 checks every virtual instruction, maximizing responsiveness as well as overhead.

Obsoleto desde a versão 3.2: This function doesn't have an effect anymore, as the internal logic for thread switching and asynchronous tasks has been rewritten. Use `setswitchinterval()` instead.

`sys.`**`setdlopenflags`**(*n*)
Set the flags used by the interpreter for `dlopen()` calls, such as when the interpreter loads extension modules. Among other things, this will enable a lazy resolving of symbols when importing a module, if called as `sys.setdlopenflags(0)`. To share symbols across extension modules, call as `sys.setdlopenflags(os.RTLD_GLOBAL)`. Symbolic names for the flag values can be found in the `os` module (`RTLD_xxx` constants, e.g. `os.RTLD_LAZY`).

Disponibilidade: Unix.

`sys.`**`set_int_max_str_digits`**(*n*)
Set the *integer string conversion length limitation* used by this interpreter. See also `get_int_max_str_digits()`.

Novo na versão 3.7.14.

`sys.`**`setprofile`**(*profilefunc*)
Set the system's profile function, which allows you to implement a Python source code profiler in Python. See chapter *The Python Profilers* for more information on the Python profiler. The system's profile function is called similarly to the system's trace function (see `settrace()`), but it is called with different events, for example it isn't called for each executed line of code (only on call and return, but the return event is reported even when an exception has been set). The function is thread-specific, but there is no way for the profiler to know about context switches between threads, so it does not make sense to use this in the presence of multiple threads. Also, its return value is not used, so it can simply return `None`. Error in the profile function will cause itself unset.

Profile functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: `'call'`, `'return'`, `'c_call'`, `'c_return'`, or `'c_exception'`. *arg* depends on the event type.

The events have the following meaning:

`'call'` A function is called (or some other code block entered). The profile function is called; *arg* is `None`.

`'return'` A function (or other code block) is about to return. The profile function is called; *arg* is the value that will be returned, or `None` if the event is caused by an exception being raised.

`'c_call'` A C function is about to be called. This may be an extension function or a built-in. *arg* is the C function object.

`'c_return'` A C function has returned. *arg* is the C function object.

`'c_exception'` A C function has raised an exception. *arg* is the C function object.

`sys.`**`setrecursionlimit`**(*limit*)
Set the maximum depth of the Python interpreter stack to *limit*. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.

The highest possible limit is platform-dependent. A user may need to set the limit higher when they have a program that requires deep recursion and a platform that supports a higher limit. This should be done with care, because a too-high limit can lead to a crash.

If the new limit is too low at the current recursion depth, a `RecursionError` exception is raised.

Alterado na versão 3.5.1: A `RecursionError` exception is now raised if the new limit is too low at the current recursion depth.

`sys.`**`setswitchinterval`**(*interval*)
Set the interpreter's thread switch interval (in seconds). This floating-point value determines the ideal duration of

the "timeslices" allocated to concurrently running Python threads. Please note that the actual value can be higher, especially if long-running internal functions or methods are used. Also, which thread becomes scheduled at the end of the interval is the operating system's decision. The interpreter doesn't have its own scheduler.

Novo na versão 3.2.

sys.**settrace**(*tracefunc*)

Set the system's trace function, which allows you to implement a Python source code debugger in Python. The function is thread-specific; for a debugger to support multiple threads, it must register a trace function using *settrace()* for each thread being debugged or use *threading.settrace()*.

Trace functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: `'call'`, `'line'`, `'return'`, `'exception'` or `'opcode'`. *arg* depends on the event type.

The trace function is invoked (with *event* set to `'call'`) whenever a new local scope is entered; it should return a reference to a local trace function to be used for the new scope, or `None` if the scope shouldn't be traced.

The local trace function should return a reference to itself (or to another function for further tracing in that scope), or `None` to turn off tracing in that scope.

If there is any error occurred in the trace function, it will be unset, just like `settrace(None)` is called.

The events have the following meaning:

`'call'` A function is called (or some other code block entered). The global trace function is called; *arg* is `None`; the return value specifies the local trace function.

`'line'` The interpreter is about to execute a new line of code or re-execute the condition of a loop. The local trace function is called; *arg* is `None`; the return value specifies the new local trace function. See `Objects/lnotab_notes.txt` for a detailed explanation of how this works. Per-line events may be disabled for a frame by setting `f_trace_lines` to *False* on that frame.

`'return'` A function (or other code block) is about to return. The local trace function is called; *arg* is the value that will be returned, or `None` if the event is caused by an exception being raised. The trace function's return value is ignored.

`'exception'` An exception has occurred. The local trace function is called; *arg* is a tuple `(exception, value, traceback)`; the return value specifies the new local trace function.

`'opcode'` The interpreter is about to execute a new opcode (see *dis* for opcode details). The local trace function is called; *arg* is `None`; the return value specifies the new local trace function. Per-opcode events are not emitted by default: they must be explicitly requested by setting `f_trace_opcodes` to *True* on the frame.

Note that as an exception is propagated down the chain of callers, an `'exception'` event is generated at each level.

For more fine-grained usage, it's possible to set a trace function by assigning `frame.f_trace = tracefunc` explicitly, rather than relying on it being set indirectly via the return value from an already installed trace function. This is also required for activating the trace function on the current frame, which *settrace()* doesn't do. Note that in order for this to work, a global tracing function must have been installed with *settrace()* in order to enable the runtime tracing machinery, but it doesn't need to be the same tracing function (e.g. it could be a low overhead tracing function that simply returns `None` to disable itself immediately on each frame).

For more information on code and frame objects, refer to types.

CPython implementation detail: The *settrace()* function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

Alterado na versão 3.7: `'opcode'` event type added; `f_trace_lines` and `f_trace_opcodes` attributes added to frames

sys.**set_asyncgen_hooks**(*firstiter*, *finalizer*)
Accepts two optional keyword arguments which are callables that accept an *asynchronous generator iterator* as an argument. The *firstiter* callable will be called when an asynchronous generator is iterated for the first time. The *finalizer* will be called when an asynchronous generator is about to be garbage collected.

Novo na versão 3.6: See **PEP 525** for more details, and for a reference example of a *finalizer* method see the implementation of `asyncio.Loop.shutdown_asyncgens` in Lib/asyncio/base_events.py

Nota: Esta função foi adicionada provisoriamente (veja **PEP 411** para detalhes.)

sys.**set_coroutine_origin_tracking_depth**(*depth*)
Allows enabling or disabling coroutine origin tracking. When enabled, the `cr_origin` attribute on coroutine objects will contain a tuple of (filename, line number, function name) tuples describing the traceback where the coroutine object was created, with the most recent call first. When disabled, `cr_origin` will be None.

To enable, pass a *depth* value greater than zero; this sets the number of frames whose information will be captured. To disable, pass set *depth* to zero.

This setting is thread-specific.

Novo na versão 3.7.

Nota: Esta função foi adicionada provisoriamente (veja **PEP 411** para detalhes.) Use-a apenas para propósitos de depuração.

sys.**set_coroutine_wrapper**(*wrapper*)
Allows intercepting creation of *coroutine* objects (only ones that are created by an `async def` function; generators decorated with *types.coroutine()* or *asyncio.coroutine()* will not be intercepted).

The *wrapper* argument must be either:

- a callable that accepts one argument (a coroutine object);
- `None`, to reset the wrapper.

If called twice, the new wrapper replaces the previous one. The function is thread-specific.

The *wrapper* callable cannot define new coroutines directly or indirectly:

```
def wrapper(coro):
    async def wrap(coro):
        return await coro
    return wrap(coro)
sys.set_coroutine_wrapper(wrapper)

async def foo():
    pass

# The following line will fail with a RuntimeError, because
# ``wrapper`` creates a ``wrap(coro)`` coroutine:
foo()
```

See also *get_coroutine_wrapper()*.

Novo na versão 3.5: See **PEP 492** for more details.

Nota: Esta função foi adicionada provisoriamente (veja **PEP 411** para detalhes.) Use-a apenas para propósitos de depuração.

Obsoleto desde a versão 3.7: The coroutine wrapper functionality has been deprecated, and will be removed in 3.8. See bpo-32591 for details.

`sys._enablelegacywindowsfsencoding()`
Changes the default filesystem encoding and errors mode to 'mbcs' and 'replace' respectively, for consistency with versions of Python prior to 3.6.

This is equivalent to defining the `PYTHONLEGACYWINDOWSFSENCODING` environment variable before launching Python.

Disponibilidade: Windows.

Novo na versão 3.6: Veja **PEP 529** para mais detalhes.

`sys.stdin`
`sys.stdout`
`sys.stderr`
File objects used by the interpreter for standard input, output and errors:

- `stdin` is used for all interactive input (including calls to *`input()`*);
- `stdout` is used for the output of *`print()`* and *expression* statements and for the prompts of *`input()`*;
- The interpreter's own prompts and its error messages go to `stderr`.

These streams are regular *text files* like those returned by the *`open()`* function. Their parameters are chosen as follows:

- The character encoding is platform-dependent. Non-Windows platforms use the locale encoding (see *`locale.getpreferredencoding()`*).

  On Windows, UTF-8 is used for the console device. Non-character devices such as disk files and pipes use the system locale encoding (i.e. the ANSI codepage). Non-console character devices such as NUL (i.e. where `isatty()` returns `True`) use the value of the console input and output codepages at startup, respectively for stdin and stdout/stderr. This defaults to the system locale encoding if the process is not initially attached to a console.

  The special behaviour of the console can be overridden by setting the environment variable PYTHONLEGACYWINDOWSSTDIO before starting Python. In that case, the console codepages are used as for any other character device.

  Under all platforms, you can override the character encoding by setting the `PYTHONIOENCODING` environment variable before starting Python or by using the new `-X utf8` command line option and `PYTHONUTF8` environment variable. However, for the Windows console, this only applies when `PYTHONLEGACYWINDOWSSTDIO` is also set.

- When interactive, `stdout` and `stderr` streams are line-buffered. Otherwise, they are block-buffered like regular text files. You can override this value with the `-u` command-line option.

Nota: To write or read binary data from/to the standard streams, use the underlying binary *`buffer`* object. For example, to write bytes to *`stdout`*, use `sys.stdout.buffer.write(b'abc')`.

However, if you are writing a library (and do not control in which context its code will be executed), be aware that the standard streams may be replaced with file-like objects like *`io.StringIO`* which do not support the `buffer` attribute.

sys.**__stdin__**
sys.**__stdout__**
sys.**__stderr__**
These objects contain the original values of `stdin`, `stderr` and `stdout` at the start of the program. They are used during finalization, and could be useful to print to the actual standard stream no matter if the `sys.std*` object has been redirected.

It can also be used to restore the actual files to known working file objects in case they have been overwritten with a broken object. However, the preferred way to do this is to explicitly save the previous stream before replacing it, and restore the saved object.

Nota: Under some conditions `stdin`, `stdout` and `stderr` as well as the original values `__stdin__`, `__stdout__` and `__stderr__` can be `None`. It is usually the case for Windows GUI apps that aren't connected to a console and Python apps started with **pythonw**.

---

sys.**thread_info**
A *named tuple* holding information about the thread implementation.

Atributo	Explanação
`name`	Name of the thread implementation: • `'nt'`: Windows threads • `'pthread'`: threads POSIX • `'solaris'`: Solaris threads
`lock`	Name of the lock implementation: • `'semaphore'`: a lock uses a semaphore • `'mutex+cond'`: a lock uses a mutex and a condition variable • `None` if this information is unknown
`version`	Name and version of the thread library. It is a string, or `None` if this information is unknown.

Novo na versão 3.3.

sys.**tracebacklimit**
When this variable is set to an integer value, it determines the maximum number of levels of traceback information printed when an unhandled exception occurs. The default is `1000`. When set to `0` or less, all traceback information is suppressed and only the exception type and value are printed.

sys.**version**
A string containing the version number of the Python interpreter plus additional information on the build number and compiler used. This string is displayed when the interactive interpreter is started. Do not extract version information out of it, rather, use *`version_info`* and the functions provided by the *`platform`* module.

sys.**api_version**
The C API version for this interpreter. Programmers may find this useful when debugging version conflicts between Python and extension modules.

sys.**version_info**
A tuple containing the five components of the version number: *major*, *minor*, *micro*, *releaselevel*, and *serial*. All values except *releaselevel* are integers; the release level is `'alpha'`, `'beta'`, `'candidate'`, or `'final'`. The `version_info` value corresponding to the Python version 2.0 is `(2, 0, 0, 'final', 0)`. The components can also be accessed by name, so `sys.version_info[0]` is equivalent to `sys.version_info.major` and so on.

Alterado na versão 3.1: Added named component attributes.

sys.**warnoptions**
This is an implementation detail of the warnings framework; do not modify this value. Refer to the *warnings* module for more information on the warnings framework.

sys.**winver**
The version number used to form registry keys on Windows platforms. This is stored as string resource 1000 in the Python DLL. The value is normally the first three characters of *version*. It is provided in the *sys* module for informational purposes; modifying this value has no effect on the registry keys used by Python.

Disponibilidade: Windows.

sys.**_xoptions**
A dictionary of the various implementation-specific flags passed through the `-X` command-line option. Option names are either mapped to their values, if given explicitly, or to *True*. Example:

```
$ ./python -Xa=b -Xc
Python 3.2a3+ (py3k, Oct 16 2010, 20:14:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys._xoptions
{'a': 'b', 'c': True}
```

CPython implementation detail: This is a CPython-specific way of accessing options passed through `-X`. Other implementations may export them through other means, or not at all.

Novo na versão 3.2.

Citations

30.2 sysconfig — Provide access to Python's configuration information

Novo na versão 3.2.

Código Fonte: Lib/sysconfig.py

The *sysconfig* module provides access to Python's configuration information like the list of installation paths and the configuration variables relevant for the current platform.

30.2.1 Configuration variables

A Python distribution contains a `Makefile` and a `pyconfig.h` header file that are necessary to build both the Python binary itself and third-party C extensions compiled using *distutils*.

sysconfig puts all variables found in these files in a dictionary that can be accessed using *get_config_vars()* or *get_config_var()*.

Notice that on Windows, it's a much smaller set.

sysconfig.**get_config_vars**(**args*)
With no arguments, return a dictionary of all configuration variables relevant for the current platform.

With arguments, return a list of values that result from looking up each argument in the configuration variable dictionary.

Para cada argumento, se o valor não for encontrado, retorna `None`.

`sysconfig.`**`get_config_var`**`(`*name*`)`
Return the value of a single variable *name*. Equivalent to `get_config_vars().get(name)`.

If *name* is not found, return `None`.

Example of usage:

```
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
```

30.2.2 Installation paths

Python uses an installation scheme that differs depending on the platform and on the installation options. These schemes are stored in *sysconfig* under unique identifiers based on the value returned by *os.name*.

Every new component that is installed using *distutils* or a Distutils-based system will follow the same scheme to copy its file in the right places.

Python currently supports seven schemes:

- *posix_prefix*: scheme for POSIX platforms like Linux or Mac OS X. This is the default scheme used when Python or a component is installed.
- *posix_home*: scheme for POSIX platforms used when a *home* option is used upon installation. This scheme is used when a component is installed through Distutils with a specific home prefix.
- *posix_user*: scheme for POSIX platforms used when a component is installed through Distutils and the *user* option is used. This scheme defines paths located under the user home directory.
- *nt*: scheme for NT platforms like Windows.
- *nt_user*: scheme for NT platforms, when the *user* option is used.

Each scheme is itself composed of a series of paths and each path has a unique identifier. Python currently uses eight paths:

- *stdlib*: directory containing the standard Python library files that are not platform-specific.
- *platstdlib*: directory containing the standard Python library files that are platform-specific.
- *platlib*: directory for site-specific, platform-specific files.
- *purelib*: directory for site-specific, non-platform-specific files.
- *include*: directory for non-platform-specific header files.
- *platinclude*: directory for platform-specific header files.
- *scripts*: directory for script files.
- *data*: directory for data files.

sysconfig provides some functions to determine these paths.

sysconfig.**get_scheme_names**()
Return a tuple containing all schemes currently supported in *sysconfig*.

sysconfig.**get_path_names**()
Return a tuple containing all path names currently supported in *sysconfig*.

sysconfig.**get_path**(*name*[, *scheme*[, *vars*[, *expand*]]])
Return an installation path corresponding to the path *name*, from the install scheme named *scheme*.

name has to be a value from the list returned by *get_path_names()*.

sysconfig stores installation paths corresponding to each path name, for each platform, with variables to be expanded. For instance the *stdlib* path for the *nt* scheme is: `{base}/Lib`.

get_path() will use the variables returned by *get_config_vars()* to expand the path. All variables have default values for each platform so one may call this function and get the default value.

If *scheme* is provided, it must be a value from the list returned by *get_scheme_names()*. Otherwise, the default scheme for the current platform is used.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary return by *get_config_vars()*.

If *expand* is set to `False`, the path will not be expanded using the variables.

If *name* is not found, return `None`.

sysconfig.**get_paths**([*scheme*[, *vars*[, *expand*]]])
Return a dictionary containing all installation paths corresponding to an installation scheme. See *get_path()* for more information.

If *scheme* is not provided, will use the default scheme for the current platform.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary used to expand the paths.

If *expand* is set to false, the paths will not be expanded.

If *scheme* is not an existing scheme, *get_paths()* will raise a *KeyError*.

30.2.3 Outras funções

sysconfig.**get_python_version**()
Return the `MAJOR.MINOR` Python version number as a string. Similar to `'%d.%d' % sys.version_info[:2]`.

sysconfig.**get_platform**()
Return a string that identifies the current platform.

This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by 'os.uname()'), although the exact information included depends on the OS; e.g., on Linux, the kernel version isn't particularly important.

Exemplos de valores retornados:

- linux-i586
- linux-alpha (?)
- solaris-2.6-sun4u

Windows will return one of:

- win-amd64 (64bit Windows on AMD64, aka x86_64, Intel64, and EM64T)

- win32 (all others - specifically, sys.platform is returned)

Mac OS X pode retornar:

- macosx-10.6-ppc
- macosx-10.4-ppc64
- macosx-10.3-i386
- macosx-10.4-fat

For other non-POSIX platforms, currently just returns *sys.platform*.

sysconfig.**is_python_build**()
Return `True` if the running Python interpreter was built from source and is being run from its built location, and not from a location resulting from e.g. running `make install` or installing via a binary installer.

sysconfig.**parse_config_h**(*fp*[, *vars*])
Parse a `config.h`-style file.

fp is a file-like object pointing to the `config.h`-like file.

A dictionary containing name/value pairs is returned. If an optional dictionary is passed in as the second argument, it is used instead of a new dictionary, and updated with the values read in the file.

sysconfig.**get_config_h_filename**()
Return the path of `pyconfig.h`.

sysconfig.**get_makefile_filename**()
Return the path of `Makefile`.

30.2.4 Usando o módulo `sysconfig` como um Script

You can use *sysconfig* as a script with Python's *-m* option:

```
$ python -m sysconfig
Platform: "macosx-10.4-i386"
Python version: "3.2"
Current installation scheme: "posix_prefix"

Paths:
        data = "/usr/local"
        include = "/Users/tarek/Dev/svn.python.org/py3k/Include"
        platinclude = "."
        platlib = "/usr/local/lib/python3.2/site-packages"
        platstdlib = "/usr/local/lib/python3.2"
        purelib = "/usr/local/lib/python3.2/site-packages"
        scripts = "/usr/local/bin"
        stdlib = "/usr/local/lib/python3.2"

Variables:
        AC_APPLE_UNIVERSAL_BUILD = "0"
        AIX_GENUINE_CPLUSPLUS = "0"
        AR = "ar"
        ARFLAGS = "rc"
        ...
```

This call will print in the standard output the information returned by *get_platform()*, *get_python_version()*, *get_path()* and *get_config_vars()*.

30.3 builtins — Objetos Built-in

Este módulo fornece acesso direto a todos os identificadores "built-in" do Python; Por exemplo, `builtins.open` é o nome completo para a função interna *open()*. Veja *Funções embutidas* e :ref:' built-in-consts' para documentação.

Este módulo normalmente não é acessado explicitamente pela maioria dos aplicativos, mas pode ser útil em módulos que fornecem objetos com o mesmo nome como um valor embutido, mas em que o objeto embutido desse nome também é necessário. Por exemplo, em um módulo que deseja implementar uma função: func: *open* que envolve o embutido: func:' open', este módulo pode ser usado diretamente

```
import builtins

def open(path):
    f = builtins.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to upper-case.'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

    # ...
```

Como um detalhe de implementação, a maioria dos módulos tem o nome __builtins__ disponibilizados como parte de seus globais. O valor de __builtins__ normalmente, este é o módulo ou o valor desse módulo *__dict__* atributo. Uma vez que este é um detalhe de implementação, ele não pode ser usado por implementações alternativas do Python.

30.4 __main__ — Ambiente de Script de Nível Superior

'__main__' é o nome do escopo no qual o código de nível mais alto executa. O *__name__* do módulo é definido como '__main__' quando for lido a partir de uma entrada padrão, um script ou uma tela de comando interativo.

Um módulo pode verificar se está ou não rodando no escopo principal, verificando seu próprio __name__, o que permite expressões para executar condicionalmente o código em um módulo quando esse executa como um script ou com `python -m` mas não quando esse é importado:

```
if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Para um pacote, o mesmo resultado pode ser obtido incluindo um módulo `__main__.py` , o conteúdo desse será executado quando rodar o módulo com `-m`

30.5 `warnings` — Controle de avisos

Código Fonte: Lib/warnings.py

As mensagens de aviso são normalmente emitidas em situações em que é útil alertar o usuário sobre alguma condição em um programa, onde essa condição (normalmente) não garante o levantamento de uma exceção e o encerramento do programa. Por exemplo, pode-se querer emitir um aviso quando um programa usa um módulo obsoleto.

Os programadores Python emitem avisos chamando a função *warn()* definida neste módulo. (Os programadores C usam `PyErr_WarnEx()`; veja exceptionhandling para detalhes).

Warning messages are normally written to `sys.stderr`, but their disposition can be changed flexibly, from ignoring all warnings to turning them into exceptions. The disposition of warnings can vary based on the warning category (see below), the text of the warning message, and the source location where it is issued. Repetitions of a particular warning for the same source location are typically suppressed.

Existem duas etapas no controle de avisos: primeiro, cada vez que um aviso é emitido, é feita uma determinação se uma mensagem deve ser emitida ou não; a seguir, se uma mensagem deve ser emitida, ela é formatada e impressa usando um gancho configurável pelo usuário.

The determination whether to issue a warning message is controlled by the warning filter, which is a sequence of matching rules and actions. Rules can be added to the filter by calling *filterwarnings()* and reset to its default state by calling *resetwarnings()*.

A exibição de mensagens de aviso é feita chamando *showwarning()*, que pode ser substituída; a implementação padrão desta função formata a mensagem chamando *formatwarning()*, que também está disponível para uso por implementações personalizadas.

Ver também:

logging.captureWarnings() permite que você manipule todos os avisos com a infraestrutura de registro padrão.

30.5.1 Categorias de avisos

Existem várias exceções embutidas que representam categorias de aviso. Essa categorização é útil para filtrar grupos de avisos.

Embora sejam tecnicamente *exceções embutidas*, elas são documentadas aqui, porque conceitualmente pertencem ao mecanismo de avisos.

O código do usuário pode definir categorias de aviso adicionais criando uma subclasse de uma das categorias de aviso padrão. Uma categoria de aviso deve ser sempre uma subclasse da classe *Warning*.

As seguintes classes de categorias de avisos estão definidas atualmente:

Classe	Description (descrição)
Warning	Esta é a classe base de todas as classes de categoria de aviso. É uma subclasse de *Exception*.
UserWarning	A categoria padrão para *warn()*.
DeprecationWarning	Categoria base para avisos sobre recursos descontinuados quando esses avisos são destinados a outros desenvolvedores Python (ignorado por padrão, a menos que acionado por código em __main__).
SyntaxWarning	Categoria base para avisos sobre recursos sintáticos duvidosos.
RuntimeWarning	Categoria base para avisos sobre recursos duvidosos de tempo de execução.
FutureWarning	Categoria base para avisos sobre recursos descontinuados quando esses avisos se destinam a usuários finais de aplicações escritas em Python.
PendingDeprecationWarning	Categoria base para avisos sobre recursos que serão descontinuados no futuro (ignorados por padrão).
ImportWarning	Categoria base para avisos acionados durante o processo de importação de um módulo (ignorado por padrão).
UnicodeWarning	Categoria base para avisos relacionados a Unicode.
BytesWarning	Categoria base para avisos relacionados a *bytes* e *bytearray*.
ResourceWarning	Categoria base para avisos relacionados a uso de recursos.

Alterado na versão 3.7: Anteriormente, *DeprecationWarning* e *FutureWarning* eram diferenciadas com base em se um recurso estava sendo removido completamente ou mudando seu comportamento. Elas agora são diferenciadas com base em seu público-alvo e na maneira como são tratadas pelos filtros de avisos padrão.

30.5.2 O filtro de avisos

O filtro de avisos controla se os avisos são ignorados, exibidos ou transformados em erros (levantando uma exceção).

Conceitualmente, o filtro de avisos mantém uma lista ordenada de especificações de filtro; qualquer aviso específico é comparado com cada especificação de filtro na lista, por sua vez, até que uma correspondência seja encontrada; o filtro determina a disposição da correspondência. Cada entrada é uma tupla no formato (*action*, *message*, *category*, *module*, *lineno*), sendo:

- *action* é uma das seguintes strings:

Valor	Disposição
"default"	exibe a primeira ocorrência de avisos correspondentes para cada local (módulo + número da linha) onde o aviso é emitido
"error"	transforma avisos correspondentes em exceções
"ignore"	nunca exibe avisos correspondentes
"always"	sempre exibe avisos correspondentes
"module"	exibe a primeira ocorrência de avisos correspondentes para cada módulo onde o aviso é emitido (independentemente do número da linha)
"once"	exibe apenas a primeira ocorrência de avisos correspondentes, independentemente da localização

- *message* é uma string que contém uma expressão regular que deve corresponder ao início da mensagem de aviso. A expressão é compilada para não fazer distinção entre maiúsculas e minúsculas.
- *category* é uma classe (uma subclasse de *Warning*) da qual a categoria de aviso deve ser uma subclasse para corresponder.
- *module* é uma string que contém uma expressão regular à qual o nome do módulo deve corresponder. A expressão é compilada para fazer distinção entre maiúsculas e minúsculas.

- *lineno* é um número inteiro que deve corresponder ao número da linha onde ocorreu o aviso, ou `0` para corresponder a todos os números de linha.

Como a classe *`Warning`* é derivada da classe embutida *`Exception`*, para transformar um aviso em um erro, simplesmente levantamos `category(message)`.

Se um aviso for relatado e não corresponder a nenhum filtro registrado, a ação "padrão" será aplicada (daí seu nome).

Describing Warning Filters

The warnings filter is initialized by `-W` options passed to the Python interpreter command line and the `PYTHONWARNINGS` environment variable. The interpreter saves the arguments for all supplied entries without interpretation in `sys.warnoptions`; the *`warnings`* module parses these when it is first imported (invalid options are ignored, after printing a message to `sys.stderr`).

Individual warnings filters are specified as a sequence of fields separated by colons:

```
action:message:category:module:line
```

The meaning of each of these fields is as described in *O filtro de avisos*. When listing multiple filters on a single line (as for `PYTHONWARNINGS`), the individual filters are separated by commas,and the filters listed later take precedence over those listed before them (as they're applied left-to-right, and the most recently applied filters take precedence over earlier ones).

Commonly used warning filters apply to either all warnings, warnings in a particular category, or warnings raised by particular modules or packages. Some examples:

```
default                       # Show all warnings (even those ignored by default)
ignore                        # Ignore all warnings
error                         # Convert all warnings to errors
error::ResourceWarning        # Treat ResourceWarning messages as errors
default::DeprecationWarning   # Show DeprecationWarning messages
ignore,default:::mymodule     # Only report warnings triggered by "mymodule"
error:::mymodule[.*]          # Convert warnings to errors in "mymodule"
                              # and any subpackages of "mymodule"
```

Filtro de avisos padrão

By default, Python installs several warning filters, which can be overridden by the `-W` command-line option, the `PYTHONWARNINGS` environment variable and calls to *`filterwarnings()`*.

In regular release builds, the default warning filter has the following entries (in order of precedence):

```
default::DeprecationWarning:__main__
ignore::DeprecationWarning
ignore::PendingDeprecationWarning
ignore::ImportWarning
ignore::ResourceWarning
```

In debug builds, the list of default warning filters is empty.

Alterado na versão 3.2: *`DeprecationWarning`* is now ignored by default in addition to *`PendingDeprecationWarning`*.

Alterado na versão 3.7: *`DeprecationWarning`* is once again shown by default when triggered directly by code in `__main__`.

Alterado na versão 3.7: *BytesWarning* no longer appears in the default filter list and is instead configured via *sys.warnoptions* when `-b` is specified twice.

Overriding the default filter

Developers of applications written in Python may wish to hide *all* Python level warnings from their users by default, and only display them when running tests or otherwise working on the application. The *sys.warnoptions* attribute used to pass filter configurations to the interpreter can be used as a marker to indicate whether or not warnings should be disabled:

```
import sys

if not sys.warnoptions:
    import warnings
    warnings.simplefilter("ignore")
```

Developers of test runners for Python code are advised to instead ensure that *all* warnings are displayed by default for the code under test, using code like:

```
import sys

if not sys.warnoptions:
    import os, warnings
    warnings.simplefilter("default") # Change the filter in this process
    os.environ["PYTHONWARNINGS"] = "default" # Also affect subprocesses
```

Finally, developers of interactive shells that run user code in a namespace other than `__main__` are advised to ensure that *DeprecationWarning* messages are made visible by default, using code like the following (where `user_ns` is the module used to execute code entered interactively):

```
import warnings
warnings.filterwarnings("default", category=DeprecationWarning,
                                   module=user_ns.get("__name__"))
```

30.5.3 Temporarily Suppressing Warnings

If you are using code that you know will raise a warning, such as a deprecated function, but do not want to see the warning (even when warnings have been explicitly configured via the command line), then it is possible to suppress the warning using the *catch_warnings* context manager:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

While within the context manager all warnings will simply be ignored. This allows you to use known-deprecated code without having to see the warning while not suppressing the warning for other code that might not be aware of its use of deprecated code. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the *catch_warnings* context manager at the same time, the behavior is undefined.

30.5.4 Testing Warnings

To test warnings raised by code, use the *catch_warnings* context manager. With it you can temporarily mutate the warnings filter to facilitate your testing. For instance, do the following to capture all raised warnings to check:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # Cause all warnings to always be triggered.
    warnings.simplefilter("always")
    # Trigger a warning.
    fxn()
    # Verify some things
    assert len(w) == 1
    assert issubclass(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

One can also cause all warnings to be exceptions by using `error` instead of `always`. One thing to be aware of is that if a warning has already been raised because of a `once`/`default` rule, then no matter what filters are set the warning will not be seen again unless the warnings registry related to the warning has been cleared.

Once the context manager exits, the warnings filter is restored to its state when the context was entered. This prevents tests from changing the warnings filter in unexpected ways between tests and leading to indeterminate test results. The *showwarning()* function in the module is also restored to its original value. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the *catch_warnings* context manager at the same time, the behavior is undefined.

When testing multiple operations that raise the same kind of warning, it is important to test them in a manner that confirms each operation is raising a new warning (e.g. set warnings to be raised as exceptions and check the operations raise exceptions, check that the length of the warning list continues to increase after each operation, or else delete the previous entries from the warnings list before each new operation).

30.5.5 Updating Code For New Versions of Dependencies

Warning categories that are primarily of interest to Python developers (rather than end users of applications written in Python) are ignored by default.

Notably, this "ignored by default" list includes *DeprecationWarning* (for every module except `__main__`), which means developers should make sure to test their code with typically ignored warnings made visible in order to receive timely notifications of future breaking API changes (whether in the standard library or third party packages).

In the ideal case, the code will have a suitable test suite, and the test runner will take care of implicitly enabling all warnings when running tests (the test runner provided by the *unittest* module does this).

In less ideal cases, applications can be checked for use of deprecated interfaces by passing `-Wd` to the Python interpreter (this is shorthand for `-W default`) or setting `PYTHONWARNINGS=default` in the environment. This enables default handling for all warnings, including those that are ignored by default. To change what action is taken for encountered warnings you can change what argument is passed to `-W` (e.g. `-W error`). See the `-W` flag for more details on what is possible.

30.5.6 Available Functions

warnings.**warn**(*message*, *category=None*, *stacklevel=1*, *source=None*)

Issue a warning, or maybe ignore it or raise an exception. The *category* argument, if given, must be a warning category class (see above); it defaults to `UserWarning`. Alternatively *message* can be a `Warning` instance, in which case *category* will be ignored and `message.__class__` will be used. In this case the message text will be `str(message)`. This function raises an exception if the particular warning issued is changed into an error by the warnings filter see above. The *stacklevel* argument can be used by wrapper functions written in Python, like this:

```
def deprecation(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

This makes the warning refer to `deprecation()`'s caller, rather than to the source of `deprecation()` itself (since the latter would defeat the purpose of the warning message).

source, if supplied, is the destroyed object which emitted a `ResourceWarning`.

Alterado na versão 3.6: Added *source* parameter.

warnings.**warn_explicit**(*message*, *category*, *filename*, *lineno*, *module=None*, *registry=None*, *module_globals=None*, *source=None*)

This is a low-level interface to the functionality of `warn()`, passing in explicitly the message, category, filename and line number, and optionally the module name and the registry (which should be the `__warningregistry__` dictionary of the module). The module name defaults to the filename with `.py` stripped; if no registry is passed, the warning is never suppressed. *message* must be a string and *category* a subclass of `Warning` or *message* may be a `Warning` instance, in which case *category* will be ignored.

module_globals, if supplied, should be the global namespace in use by the code for which the warning is issued. (This argument is used to support displaying source for modules found in zipfiles or other non-filesystem import sources).

source, if supplied, is the destroyed object which emitted a `ResourceWarning`.

Alterado na versão 3.6: Add the *source* parameter.

warnings.**showwarning**(*message*, *category*, *filename*, *lineno*, *file=None*, *line=None*)

Write a warning to a file. The default implementation calls `formatwarning(message, category, filename, lineno, line)` and writes the resulting string to *file*, which defaults to `sys.stderr`. You may replace this function with any callable by assigning to `warnings.showwarning`. *line* is a line of source code to be included in the warning message; if *line* is not supplied, `showwarning()` will try to read the line specified by *filename* and *lineno*.

warnings.**formatwarning**(*message*, *category*, *filename*, *lineno*, *line=None*)

Format a warning the standard way. This returns a string which may contain embedded newlines and ends in a newline. *line* is a line of source code to be included in the warning message; if *line* is not supplied, `formatwarning()` will try to read the line specified by *filename* and *lineno*.

warnings.**filterwarnings**(*action*, *message=''*, *category=Warning*, *module=''*, *lineno=0*, *append=False*)

Insert an entry into the list of *warnings filter specifications*. The entry is inserted at the front by default; if *append* is true, it is inserted at the end. This checks the types of the arguments, compiles the *message* and *module* regular expressions, and inserts them as a tuple in the list of warnings filters. Entries closer to the front of the list override entries later in the list, if both match a particular warning. Omitted arguments default to a value that matches everything.

warnings.**simplefilter**(*action*, *category=Warning*, *lineno=0*, *append=False*)

Insert a simple entry into the list of *warnings filter specifications*. The meaning of the function parameters is as for `filterwarnings()`, but regular expressions are not needed as the filter inserted always matches any message in any module as long as the category and line number match.

warnings.**resetwarnings**()
Reset the warnings filter. This discards the effect of all previous calls to *filterwarnings()*, including that of the -W command line options and calls to *simplefilter()*.

30.5.7 Available Context Managers

class warnings.**catch_warnings**(*, *record=False*, *module=None*)
A context manager that copies and, upon exit, restores the warnings filter and the *showwarning()* function. If the *record* argument is *False* (the default) the context manager returns *None* on entry. If *record* is *True*, a list is returned that is progressively populated with objects as seen by a custom *showwarning()* function (which also suppresses output to sys.stdout). Each object in the list has attributes with the same names as the arguments to *showwarning()*.

The *module* argument takes a module that will be used instead of the module returned when you import *warnings* whose filter will be protected. This argument exists primarily for testing the *warnings* module itself.

Nota: The *catch_warnings* manager works by replacing and then later restoring the module's *showwarning()* function and internal list of filter specifications. This means the context manager is modifying global state and therefore is not thread-safe.

30.6 dataclasses — Data Classes

Código fonte: Lib/dataclasses.py

Este módulo fornece um decorador e funções para adicionar automaticamente *método especial*s tais como __init__() e __repr__() a classes definidas pelo usuário. Foi originalmente descrita em **PEP 557**.

As variáveis de membro a serem usadas nesses métodos gerados são definidas usando **PEP 526** anotações de tipo. Por exemplo, este código:

```
from dataclasses import dataclass

@dataclass
class InventoryItem:
    '''Class for keeping track of an item in inventory.'''
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

Will add, among other things, a __init__() that looks like:: Vai adicionar, além de outras coisas, o __init__() que parece com:

```
def __init__(self, name: str, unit_price: float, quantity_on_hand: int=0):
    self.name = name
    self.unit_price = unit_price
    self.quantity_on_hand = quantity_on_hand
```

Observe que este método é adicionado automaticamente à classe: ele não é especificado diretamente na definição `InventoryItem` mostrada acima.

Novo na versão 3.7.

30.6.1 Decoradores no nível do módulo, classes e funções.

`@dataclasses.dataclass(*, init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False)`

Esta função é um *decorador* que é usado para adicionar *método especial*s para classes, conforme descrito abaixo.

O decorador *dataclass()* examina a classe para encontrar `field`s. Um `field` é definido como uma variável de classe que possui uma *anotação de tipo*. Com duas exceções descritas abaixo, nada em *dataclass()* examina o tipo especificado na anotação de variável.

A ordem dos campos em todos os métodos gerados é a ordem em que eles aparecem na definição de classe.

O decorador *dataclass()* adicionará vários métodos "dunder" à classe, descritos abaixo. Se algum dos métodos adicionados já existir na classe, o comportamento dependerá do parâmetro, conforme documentado abaixo. O decorador retorna a mesma classe que é chamado; nenhuma nova classe é criada.

Se *dataclass()* for usado apenas como um simples decorador, sem parâmetros, ele age como se tivesse os valores padrão documentados nessa assinatura. Ou seja, esses três usos de *dataclass()* são equivalentes:

```
@dataclass
class C:
    ...

@dataclass()
class C:
    ...

@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False,␣
↪frozen=False)
class C:
   ...
```

Os parâmetros do *dataclass()* são:

- `init`: Se verdadeiro (o padrão), o método `__init__()` será gerado.

  If the class already defines `__init__()`, this parameter is ignored. Se a classe já tenha `__init__()` definido, esse parâmetro é ignorado.

- `repr`: Se verdadeiro (o padrão), um método `__repr__()` será gerado. A sequência de string repr gerada terá o nome da classe e o nome e repr de cada campo, na ordem em que são definidos na classe. Os campos marcados como excluídos do repr não são incluidos. Por exemplo: `InventoryItem(name='widget', unit_price=3.0, quantity_on_hand=10)`.

  Se a classe já tenha `__repr__()` definido, esse parâmetro é ignorado.

- `eq`: Se verdadeiro (o padrão), um método `__eq__()` será gerado. Este método compara a classe como se fosse uma tupla de campos, em ordem. Ambas as instâncias na comparação devem ser de tipo idêntico.

  Se a classe já tenha `__eq__()` definido, esse parâmetro é ignorado.

- `order`: Se verdadeiro (o padrão é `False`), os métodos `__lt__()`, `__le__()`, `__gt__()`, e `__ge__()` serão gerados. Comparam a classe como se fosse uma tupla de campos, em ordem. Ambas instâncias na comparação devem ser de tipo idêntico. Se `order` é verdadeiro e `eq` é falso, a exceção *ValueError* é levantada.

Se a classe já define algum entre `__lt__()`, `__le__()`, `__gt__()` ou `__ge__()`, então *TypeError* é levantada.

- `unsafe_hash`: Se `False` (o padrão), um método `__hash__()` é gerado, conforme `eq` e `frozen` estão configurados.

  `__hash__()` é usado para prover o método *hash()*, e quando objetos são adicionados a coleções do tipo dicionário ou conjunto. Ter um método `__hash__()` implica que instâncias da classe serão imutáveis. Mutabilidade é uma propriedade complicada, que depende da intenção do programador, da existência e comportamento do método `__eq__()`, e dos valores dos parâmetros `eq` e `frozen` no decorador *dataclass()*.

  Por padrão, *dataclass()* não vai adicionar implicitamente um método `__hash__()`, a menos que seja seguro fazê-lo. Nem irá adicionar ou modificar um método `__hash__()` existente, definido explicitamente. Configurar o atributo de classe `__hash__ = None` tem um significado específico para o Python, conforme descrito na documentação do `__hash__()`.

  If `__hash__()` is not explicit defined, or if it is set to `None`, then *dataclass()* *may* add an implicit `__hash__()` method. Although not recommended, you can force *dataclass()* to create a `__hash__()` method with `unsafe_hash=True`. This might be the case if your class is logically immutable but can nonetheless be mutated. This is a specialized use case and should be considered carefully.

  Essas são as regras governando a criação implícita de um método `__hash__()`. Observe que não pode ter um método `__hash__()` explícito na dataclass e configurar `unsafe_hash=True`; isso resultará em um *TypeError*.

  Se `eq` e `frozen` são ambos verdadeiros, por padrão *dataclass()* vai gerar um método `__hash__()`. Se `eq` é verdadeiro e `frozen` é falso, `__hash__()` será configurado para `None`, marcando a classe como não hasheável (já que é mutável). Se `eq` é falso, `__hash__()` será deixado intocado, o que significa que o método `__hash__()` da superclasse será usado (se a superclasse é *object*, significa que voltará para o hash baseado em id).

- `frozen`: Se verdadeiro (o padrão é `False`), atribuições para os campos vão gerar uma exceção. Imita instâncias congeladas, somente leitura. Se `__setattr__()` ou `__delattr__()` é definido na classe, a exceção *TypeError* é levantada. Veja a discussão abaixo.

`fields` pode opcionalmente especificar um valor padrão, usando sintaxe Python normal:

```
@dataclass
class C:
    a: int       # 'a' has no default value
    b: int = 0   # assign a default value for 'b'
```

Nesse exemplo, `a` e `b` serão incluídos no método `__init__()` adicionado, que será definido como:

```
def __init__(self, a: int, b: int = 0):
```

TypeError will be raised if a field without a default value follows a field with a default value. This is true either when this occurs in a single class, or as a result of class inheritance.

dataclasses.**field**(*, *default=MISSING*, *default_factory=MISSING*, *repr=True*, *hash=None*, *init=True*, *compare=True*, *metadata=None*)

Para casos de uso comuns e simples, nenhuma outra funcionalidade é necessária. Existem, no entanto, alguns recursos que requerem informações adicionais por campo. Para satisfazer essa necessidade de informações adicionais, você pode substituir o valor do campo padrão por uma chamada para a função *field()* fornecida. Por exemplo:

```
@dataclass
class C:
```

(continua na próxima página)

(continuação da página anterior)

```
    mylist: List[int] = field(default_factory=list)

c = C()
c.mylist += [1, 2, 3]
```

Como mostrado acima, o valor `MISSING` é um objeto sentinela usado para detectar se os parâmetros `default` e `default_factory` são fornecidos. Este sentinela é usado porque `None` é um valor válido para `default`. Nenhum código deve usar diretamente o valor `MISSING`.

Os parâmetros do func:*field* são:

- `default`: If provided, this will be the default value for this field. This is needed because the *field()* call itself replaces the normal position of the default value.
- `default_factory`: If provided, it must be a zero-argument callable that will be called when a default value is needed for this field. Among other purposes, this can be used to specify fields with mutable default values, as discussed below. It is an error to specify both `default` and `default_factory`.
- `init`: If true (the default), this field is included as a parameter to the generated `__init__()` method.
- `repr`: If true (the default), this field is included in the string returned by the generated `__repr__()` method.
- `compare`: If true (the default), this field is included in the generated equality and comparison methods (`__eq__()`, `__gt__()`, et al.).
- `hash`: This can be a bool or `None`. If true, this field is included in the generated `__hash__()` method. If `None` (the default), use the value of `compare`: this would normally be the expected behavior. A field should be considered in the hash if it's used for comparisons. Setting this value to anything other than `None` is discouraged.

  One possible reason to set `hash=False` but `compare=True` would be if a field is expensive to compute a hash value for, that field is needed for equality testing, and there are other fields that contribute to the type's hash value. Even if a field is excluded from the hash, it will still be used for comparisons.
- `metadata`: This can be a mapping or None. None is treated as an empty dict. This value is wrapped in *MappingProxyType()* to make it read-only, and exposed on the *Field* object. It is not used at all by Data Classes, and is provided as a third-party extension mechanism. Multiple third-parties can each have their own key, to use as a namespace in the metadata.

If the default value of a field is specified by a call to *field()*, then the class attribute for this field will be replaced by the specified `default` value. If no `default` is provided, then the class attribute will be deleted. The intent is that after the *dataclass()* decorator runs, the class attributes will all contain the default values for the fields, just as if the default value itself were specified. For example, after:

```
@dataclass
class C:
    x: int
    y: int = field(repr=False)
    z: int = field(repr=False, default=10)
    t: int = 20
```

The class attribute `C.z` will be `10`, the class attribute `C.t` will be `20`, and the class attributes `C.x` and `C.y` will not be set.

class dataclasses.**Field**

Field objects describe each defined field. These objects are created internally, and are returned by the *fields()* module-level method (see below). Users should never instantiate a *Field* object directly. Its documented attributes are:

- `name`: O nome do campo.
- `type`: O tipo do campo.
- `default`, `default_factory`, `init`, `repr`, `hash`, `compare`, and `metadata` have the identical meaning and values as they do in the *`field()`* declaration.

Other attributes may exist, but they are private and must not be inspected or relied on.

`dataclasses.`**`fields`**(*class_or_instance*)

Returns a tuple of *`Field`* objects that define the fields for this dataclass. Accepts either a dataclass, or an instance of a dataclass. Raises *`TypeError`* if not passed a dataclass or instance of one. Does not return pseudo-fields which are `ClassVar` or `InitVar`.

`dataclasses.`**`asdict`**(*instance*, *, *dict_factory=dict*)

Converts the dataclass `instance` to a dict (by using the factory function `dict_factory`). Each dataclass is converted to a dict of its fields, as `name: value` pairs. dataclasses, dicts, lists, and tuples are recursed into. For example:

```
@dataclass
class Point:
     x: int
     y: int

@dataclass
class C:
     mylist: List[Point]

p = Point(10, 20)
assert asdict(p) == {'x': 10, 'y': 20}

c = C([Point(0, 0), Point(10, 4)])
assert asdict(c) == {'mylist': [{'x': 0, 'y': 0}, {'x': 10, 'y': 4}]}
```

Raises *`TypeError`* if `instance` is not a dataclass instance.

`dataclasses.`**`astuple`**(*instance*, *, *tuple_factory=tuple*)

Converts the dataclass `instance` to a tuple (by using the factory function `tuple_factory`). Each dataclass is converted to a tuple of its field values. dataclasses, dicts, lists, and tuples are recursed into.

Continuando a partir do exemplo anterior:

```
assert astuple(p) == (10, 20)
assert astuple(c) == ([(0, 0), (10, 4)],)
```

Raises *`TypeError`* if `instance` is not a dataclass instance.

`dataclasses.`**`make_dataclass`**(*cls_name*, *fields*, *, *bases=()*, *namespace=None*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*)

Creates a new dataclass with name `cls_name`, fields as defined in `fields`, base classes as given in `bases`, and initialized with a namespace as given in `namespace`. `fields` is an iterable whose elements are each either `name`, `(name, type)`, or `(name, type, Field)`. If just `name` is supplied, `typing.Any` is used for `type`. The values of `init`, `repr`, `eq`, `order`, `unsafe_hash`, and `frozen` have the same meaning as they do in *`dataclass()`*.

This function is not strictly required, because any Python mechanism for creating a new class with `__annotations__` can then apply the *`dataclass()`* function to convert that class to a dataclass. This function is provided as a convenience. For example:

```
C = make_dataclass('C',
                   [('x', int),
                     'y',
                    ('z', int, field(default=5))],
                   namespace={'add_one': lambda self: self.x + 1})
```

É equivalente a:

```
@dataclass
class C:
    x: int
    y: 'typing.Any'
    z: int = 5

    def add_one(self):
        return self.x + 1
```

dataclasses.**replace**(*instance*, ***changes*)

Creates a new object of the same type of `instance`, replacing fields with values from `changes`. If `instance` is not a Data Class, raises *`TypeError`*. If values in `changes` do not specify fields, raises *`TypeError`*.

The newly returned object is created by calling the `__init__()` method of the dataclass. This ensures that `__post_init__()`, if present, is also called.

Init-only variables without default values, if any exist, must be specified on the call to *`replace()`* so that they can be passed to `__init__()` and `__post_init__()`.

It is an error for `changes` to contain any fields that are defined as having `init=False`. A *`ValueError`* will be raised in this case.

Be forewarned about how `init=False` fields work during a call to *`replace()`*. They are not copied from the source object, but rather are initialized in `__post_init__()`, if they're initialized at all. It is expected that `init=False` fields will be rarely and judiciously used. If they are used, it might be wise to have alternate class constructors, or perhaps a custom `replace()` (or similarly named) method which handles instance copying.

dataclasses.**is_dataclass**(*class_or_instance*)

Return `True` if its parameter is a dataclass or an instance of one, otherwise return `False`.

Se você precisa saber se a classe é uma instância de dataclass (e não a dataclass de fato), então adicione uma verificação para `not isinstance(obj, type)`:

```
def is_dataclass_instance(obj):
    return is_dataclass(obj) and not isinstance(obj, type)
```

30.6.2 Processing post-init

The generated `__init__()` code will call a method named `__post_init__()`, if `__post_init__()` is defined on the class. It will normally be called as `self.__post_init__()`. However, if any `InitVar` fields are defined, they will also be passed to `__post_init__()` in the order they were defined in the class. If no `__init__()` method is generated, then `__post_init__()` will not automatically be called.

Among other uses, this allows for initializing field values that depend on one or more other fields. For example:

```
@dataclass
class C:
    a: float
    b: float
```

(continua na próxima página)

(continuação da página anterior)

```
    c: float = field(init=False)

    def __post_init__(self):
        self.c = self.a + self.b
```

See the section below on init-only variables for ways to pass parameters to `__post_init__()`. Also see the warning about how *`replace()`* handles `init=False` fields.

30.6.3 Variáveis de classe

One of two places where *`dataclass()`* actually inspects the type of a field is to determine if a field is a class variable as defined in **PEP 526**. It does this by checking if the type of the field is `typing.ClassVar`. If a field is a `ClassVar`, it is excluded from consideration as a field and is ignored by the dataclass mechanisms. Such `ClassVar` pseudo-fields are not returned by the module-level *`fields()`* function.

30.6.4 Variáveis de inicialização apenas

The other place where *`dataclass()`* inspects a type annotation is to determine if a field is an init-only variable. It does this by seeing if the type of a field is of type `dataclasses.InitVar`. If a field is an `InitVar`, it is considered a pseudo-field called an init-only field. As it is not a true field, it is not returned by the module-level *`fields()`* function. Init-only fields are added as parameters to the generated `__init__()` method, and are passed to the optional `__post_init__()` method. They are not otherwise used by dataclasses.

For example, suppose a field will be initialized from a database, if a value is not provided when creating the class:

```
@dataclass
class C:
    i: int
    j: int = None
    database: InitVar[DatabaseType] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

In this case, *`fields()`* will return *`Field`* objects for `i` and `j`, but not for `database`.

30.6.5 Frozen instances

It is not possible to create truly immutable Python objects. However, by passing `frozen=True` to the *`dataclass()`* decorator you can emulate immutability. In that case, dataclasses will add `__setattr__()` and `__delattr__()` methods to the class. These methods will raise a *`FrozenInstanceError`* when invoked.

There is a tiny performance penalty when using `frozen=True`: `__init__()` cannot use simple assignment to initialize fields, and must use `object.__setattr__()`.

30.6.6 Herança

When the dataclass is being created by the *dataclass()* decorator, it looks through all of the class's base classes in reverse MRO (that is, starting at *object*) and, for each dataclass that it finds, adds the fields from that base class to an ordered mapping of fields. After all of the base class fields are added, it adds its own fields to the ordered mapping. All of the generated methods will use this combined, calculated ordered mapping of fields. Because the fields are in insertion order, derived classes override base classes. An example:

```
@dataclass
class Base:
    x: Any = 15.0
    y: int = 0

@dataclass
class C(Base):
    z: int = 10
    x: int = 15
```

A lista final de campos é, em ordem, `x`, `y`, `z`. O tipo final de `x` é `int`, conforme especificado na classe `C`.

O método `__init__()` gerado para `C` vai se parecer com:

```
def __init__(self, x: int = 15, y: int = 0, z: int = 10):
```

30.6.7 Funções padrão de fábrica

If a *field()* specifies a `default_factory`, it is called with zero arguments when a default value for the field is needed. For example, to create a new instance of a list, use:

```
mylist: list = field(default_factory=list)
```

If a field is excluded from `__init__()` (using `init=False`) and the field also specifies `default_factory`, then the default factory function will always be called from the generated `__init__()` function. This happens because there is no other way to give the field an initial value.

30.6.8 Valores padrão mutáveis

Python stores default member variable values in class attributes. Consider this example, not using dataclasses:

```
class C:
    x = []
    def add(self, element):
        self.x.append(element)

o1 = C()
o2 = C()
o1.add(1)
o2.add(2)
assert o1.x == [1, 2]
assert o1.x is o2.x
```

Note that the two instances of class `C` share the same class variable `x`, as expected.

Usando dataclasses, *se* este código fosse válido:

```
@dataclass
class D:
    x: List = []
    def add(self, element):
        self.x += element
```

Geraria código similar a:

```
class D:
    x = []
    def __init__(self, x=x):
        self.x = x
    def add(self, element):
        self.x += element

assert D().x is D().x
```

This has the same issue as the original example using class `C`. That is, two instances of class `D` that do not specify a value for `x` when creating a class instance will share the same copy of `x`. Because dataclasses just use normal Python class creation they also share this behavior. There is no general way for Data Classes to detect this condition. Instead, dataclasses will raise a *TypeError* if it detects a default parameter of type `list`, `dict`, or `set`. This is a partial solution, but it does protect against many common errors.

Using default factory functions is a way to create new instances of mutable types as default values for fields:

```
@dataclass
class D:
    x: list = field(default_factory=list)

assert D().x is not D().x
```

30.6.9 Exceções

exception dataclasses.**FrozenInstanceError**
Raised when an implicitly defined `__setattr__()` or `__delattr__()` is called on a dataclass which was defined with `frozen=True`.

30.7 `contextlib` — Utilities for `with`-statement contexts

Código Fonte: Lib/contextlib.py

This module provides utilities for common tasks involving the `with` statement. For more information see also *Tipos de Gerenciador de Contexto* and context-managers.

30.7.1 Utilitários

Functions and classes provided:

class contextlib.**AbstractContextManager**

An *abstract base class* for classes that implement `object.__enter__()` and `object.__exit__()`. A default implementation for `object.__enter__()` is provided which returns `self` while `object.__exit__()` is an abstract method which by default returns `None`. See also the definition of *Tipos de Gerenciador de Contexto*.

Novo na versão 3.6.

class contextlib.**AbstractAsyncContextManager**

An *abstract base class* for classes that implement `object.__aenter__()` and `object.__aexit__()`. A default implementation for `object.__aenter__()` is provided which returns `self` while `object.__aexit__()` is an abstract method which by default returns `None`. See also the definition of async-context-managers.

Novo na versão 3.7.

@contextlib.**contextmanager**

This function is a *decorator* that can be used to define a factory function for `with` statement context managers, without needing to create a class or separate `__enter__()` and `__exit__()` methods.

While many objects natively support use in with statements, sometimes a resource needs to be managed that isn't a context manager in its own right, and doesn't implement a `close()` method for use with `contextlib.closing`

An abstract example would be the following to ensure correct resource management:

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwds):
    # Code to acquire resource, e.g.:
    resource = acquire_resource(*args, **kwds)
    try:
        yield resource
    finally:
        # Code to release resource, e.g.:
        release_resource(resource)

>>> with managed_resource(timeout=3600) as resource:
...     # Resource is released at the end of this block,
...     # even if code in the block raises an exception
```

The function being decorated must return a *generator*-iterator when called. This iterator must yield exactly one value, which will be bound to the targets in the `with` statement's `as` clause, if any.

At the point where the generator yields, the block nested in the `with` statement is executed. The generator is then resumed after the block is exited. If an unhandled exception occurs in the block, it is reraised inside the generator at the point where the yield occurred. Thus, you can use a `try...except...finally` statement to trap the error (if any), or ensure that some cleanup takes place. If an exception is trapped merely in order to log it or to perform some action (rather than to suppress it entirely), the generator must reraise that exception. Otherwise the generator context manager will indicate to the `with` statement that the exception has been handled, and execution will resume with the statement immediately following the `with` statement.

`contextmanager()` uses *`ContextDecorator`* so the context managers it creates can be used as decorators as well as in `with` statements. When used as a decorator, a new generator instance is implicitly created on each

function call (this allows the otherwise "one-shot" context managers created by *contextmanager()* to meet the requirement that context managers support multiple invocations in order to be used as decorators).

Alterado na versão 3.2: Use of *ContextDecorator*.

@contextlib.**asynccontextmanager**

Similar to *contextmanager()*, but creates an asynchronous context manager.

This function is a *decorator* that can be used to define a factory function for `async with` statement asynchronous context managers, without needing to create a class or separate `__aenter__()` and `__aexit__()` methods. It must be applied to an *asynchronous generator* function.

Um exemplo simples:

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def get_connection():
    conn = await acquire_db_connection()
    try:
        yield conn
    finally:
        await release_db_connection(conn)

async def get_all_users():
    async with get_connection() as conn:
        return conn.query('SELECT ...')
```

Novo na versão 3.7.

contextlib.**closing**(*thing*)

Return a context manager that closes *thing* upon completion of the block. This is basically equivalent to:

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

And lets you write code like this:

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('http://www.python.org')) as page:
    for line in page:
        print(line)
```

without needing to explicitly close `page`. Even if an error occurs, `page.close()` will be called when the `with` block is exited.

contextlib.**nullcontext**(*enter_result=None*)

Return a context manager that returns *enter_result* from `__enter__`, but otherwise does nothing. It is intended to be used as a stand-in for an optional context manager, for example:

```
def myfunction(arg, ignore_exceptions=False):
    if ignore_exceptions:
        # Use suppress to ignore all exceptions.
        cm = contextlib.suppress(Exception)
    else:
        # Do not ignore any exceptions, cm has no effect.
        cm = contextlib.nullcontext()
    with cm:
        # Do something
```

An example using *enter_result*:

```
def process_file(file_or_path):
    if isinstance(file_or_path, str):
        # If string, open file
        cm = open(file_or_path)
    else:
        # Caller is responsible for closing file
        cm = nullcontext(file_or_path)

    with cm as file:
        # Perform processing on the file
```

Novo na versão 3.7.

contextlib.**suppress**(**exceptions*)

Return a context manager that suppresses any of the specified exceptions if they occur in the body of a with statement and then resumes execution with the first statement following the end of the with statement.

As with any other mechanism that completely suppresses exceptions, this context manager should be used only to cover very specific errors where silently continuing with program execution is known to be the right thing to do.

Por exemplo:

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove('somefile.tmp')

with suppress(FileNotFoundError):
    os.remove('someotherfile.tmp')
```

This code is equivalent to:

```
try:
    os.remove('somefile.tmp')
except FileNotFoundError:
    pass

try:
    os.remove('someotherfile.tmp')
except FileNotFoundError:
    pass
```

This context manager is *reentrant*.

Novo na versão 3.4.

contextlib.**redirect_stdout**(*new_target*)

Context manager for temporarily redirecting `sys.stdout` to another file or file-like object.

This tool adds flexibility to existing functions or classes whose output is hardwired to stdout.

For example, the output of `help()` normally is sent to *sys.stdout*. You can capture that output in a string by redirecting the output to an `io.StringIO` object:

```
f = io.StringIO()
with redirect_stdout(f):
    help(pow)
s = f.getvalue()
```

To send the output of `help()` to a file on disk, redirect the output to a regular file:

```
with open('help.txt', 'w') as f:
    with redirect_stdout(f):
        help(pow)
```

To send the output of `help()` to *sys.stderr*:

```
with redirect_stdout(sys.stderr):
    help(pow)
```

Note that the global side effect on `sys.stdout` means that this context manager is not suitable for use in library code and most threaded applications. It also has no effect on the output of subprocesses. However, it is still a useful approach for many utility scripts.

This context manager is *reentrant*.

Novo na versão 3.4.

contextlib.**redirect_stderr**(*new_target*)

Similar to `redirect_stdout()` but redirecting `sys.stderr` to another file or file-like object.

This context manager is *reentrant*.

Novo na versão 3.5.

class contextlib.**ContextDecorator**

A base class that enables a context manager to also be used as a decorator.

Context managers inheriting from `ContextDecorator` have to implement `__enter__` and `__exit__` as normal. `__exit__` retains its optional exception handling even when used as a decorator.

`ContextDecorator` is used by `contextmanager()`, so you get this functionality automatically.

Example of `ContextDecorator`:

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False

>>> @mycontext()
... def function():
...     print('The bit in the middle')
...
```

(continua na próxima página)

(continuação da página anterior)

```
>>> function()
Starting
The bit in the middle
Finishing

>>> with mycontext():
...     print('The bit in the middle')
...
Starting
The bit in the middle
Finishing
```

This change is just syntactic sugar for any construct of the following form:

```
def f():
    with cm():
        # Do stuff
```

`ContextDecorator` lets you instead write:

```
@cm()
def f():
    # Do stuff
```

It makes it clear that the `cm` applies to the whole function, rather than just a piece of it (and saving an indentation level is nice, too).

Existing context managers that already have a base class can be extended by using `ContextDecorator` as a mixin class:

```
from contextlib import ContextDecorator

class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self

    def __exit__(self, *exc):
        return False
```

Nota: As the decorated function must be able to be called multiple times, the underlying context manager must support use in multiple `with` statements. If this is not the case, then the original construct with the explicit `with` statement inside the function should be used.

Novo na versão 3.2.

class contextlib.**ExitStack**

A context manager that is designed to make it easy to programmatically combine other context managers and cleanup functions, especially those that are optional or otherwise driven by input data.

For example, a set of files may easily be handled in a single with statement as follows:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # All opened files will automatically be closed at the end of
    # the with statement, even if attempts to open files later
    # in the list raise an exception
```

Each instance maintains a stack of registered callbacks that are called in reverse order when the instance is closed (either explicitly or implicitly at the end of a `with` statement). Note that callbacks are *not* invoked implicitly when the context stack instance is garbage collected.

This stack model is used so that context managers that acquire their resources in their `__init__` method (such as file objects) can be handled correctly.

Since registered callbacks are invoked in the reverse order of registration, this ends up behaving as if multiple nested `with` statements had been used with the registered set of callbacks. This even extends to exception handling - if an inner callback suppresses or replaces an exception, then outer callbacks will be passed arguments based on that updated state.

This is a relatively low level API that takes care of the details of correctly unwinding the stack of exit callbacks. It provides a suitable foundation for higher level context managers that manipulate the exit stack in application specific ways.

Novo na versão 3.3.

enter_context(*cm*)
Enters a new context manager and adds its `__exit__()` method to the callback stack. The return value is the result of the context manager's own `__enter__()` method.

These context managers may suppress exceptions just as they normally would if used directly as part of a `with` statement.

push(*exit*)
Adds a context manager's `__exit__()` method to the callback stack.

As `__enter__` is *not* invoked, this method can be used to cover part of an `__enter__()` implementation with a context manager's own `__exit__()` method.

If passed an object that is not a context manager, this method assumes it is a callback with the same signature as a context manager's `__exit__()` method and adds it directly to the callback stack.

By returning true values, these callbacks can suppress exceptions the same way context manager `__exit__()` methods can.

The passed in object is returned from the function, allowing this method to be used as a function decorator.

callback(*callback*, **args*, ***kwds*)
Accepts an arbitrary callback function and arguments and adds it to the callback stack.

Unlike the other methods, callbacks added this way cannot suppress exceptions (as they are never passed the exception details).

The passed in callback is returned from the function, allowing this method to be used as a function decorator.

pop_all()
Transfers the callback stack to a fresh *ExitStack* instance and returns it. No callbacks are invoked by this operation - instead, they will now be invoked when the new stack is closed (either explicitly or implicitly at the end of a `with` statement).

For example, a group of files can be opened as an "all or nothing" operation as follows:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # Hold onto the close method, but don't call it yet.
    close_files = stack.pop_all().close
    # If opening any file fails, all previously opened files will be
    # closed automatically. If all files are opened successfully,
    # they will remain open even after the with statement ends.
    # close_files() can then be invoked explicitly to close them all.
```

close()
Immediately unwinds the callback stack, invoking callbacks in the reverse order of registration. For any context managers and exit callbacks registered, the arguments passed in will indicate that no exception occurred.

class contextlib.**AsyncExitStack**
An asynchronous context manager, similar to *ExitStack*, that supports combining both synchronous and asynchronous context managers, as well as having coroutines for cleanup logic.

The close() method is not implemented, *aclose()* must be used instead.

enter_async_context(*cm*)
Similar to enter_context() but expects an asynchronous context manager.

push_async_exit(*exit*)
Similar to push() but expects either an asynchronous context manager or a coroutine function.

push_async_callback(*callback*, **args*, ***kwds*)
Similar to callback() but expects a coroutine function.

aclose()
Similar to close() but properly handles awaitables.

Continuing the example for *asynccontextmanager()*:

```
async with AsyncExitStack() as stack:
    connections = [await stack.enter_async_context(get_connection())
        for i in range(5)]
    # All opened connections will automatically be released at the end of
    # the async with statement, even if attempts to open a connection
    # later in the list raise an exception.
```

Novo na versão 3.7.

30.7.2 Exemplos e receitas

This section describes some examples and recipes for making effective use of the tools provided by *contextlib*.

Supporting a variable number of context managers

The primary use case for *ExitStack* is the one given in the class documentation: supporting a variable number of context managers and other cleanup operations in a single with statement. The variability may come from the number of context managers needed being driven by user input (such as opening a user specified collection of files), or from some of the context managers being optional:

```
with ExitStack() as stack:
    for resource in resources:
        stack.enter_context(resource)
    if need_special_resource():
        special = acquire_special_resource()
        stack.callback(release_special_resource, special)
    # Perform operations that use the acquired resources
```

As shown, *ExitStack* also makes it quite easy to use with statements to manage arbitrary resources that don't natively support the context management protocol.

Catching exceptions from __enter__ methods

It is occasionally desirable to catch exceptions from an __enter__ method implementation, *without* inadvertently catching exceptions from the with statement body or the context manager's __exit__ method. By using *ExitStack* the steps in the context management protocol can be separated slightly in order to allow this:

```
stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
    # handle __enter__ exception
else:
    with stack:
        # Handle normal case
```

Actually needing to do this is likely to indicate that the underlying API should be providing a direct resource management interface for use with try/except/finally statements, but not all APIs are well designed in that regard. When a context manager is the only resource management API provided, then *ExitStack* can make it easier to handle various situations that can't be handled directly in a with statement.

Cleaning up in an __enter__ implementation

As noted in the documentation of *ExitStack.push()*, this method can be useful in cleaning up an already allocated resource if later steps in the __enter__() implementation fail.

Here's an example of doing this for a context manager that accepts resource acquisition and release functions, along with an optional validation function, and maps them to the context management protocol:

```
from contextlib import contextmanager, AbstractContextManager, ExitStack

class ResourceManager(AbstractContextManager):

    def __init__(self, acquire_resource, release_resource, check_resource_ok=None):
        self.acquire_resource = acquire_resource
        self.release_resource = release_resource
        if check_resource_ok is None:
            def check_resource_ok(resource):
                return True
        self.check_resource_ok = check_resource_ok

    @contextmanager
    def _cleanup_on_error(self):
        with ExitStack() as stack:
            stack.push(self)
            yield
            # The validation check passed and didn't raise an exception
            # Accordingly, we want to keep the resource, and pass it
            # back to our caller
            stack.pop_all()

    def __enter__(self):
        resource = self.acquire_resource()
        with self._cleanup_on_error():
            if not self.check_resource_ok(resource):
                msg = "Failed validation for {!r}"
                raise RuntimeError(msg.format(resource))
```

(continua na próxima página)

(continuação da página anterior)

```
        return resource

    def __exit__(self, *exc_details):
        # We don't need to duplicate any of our resource release logic
        self.release_resource()
```

Replacing any use of `try-finally` and flag variables

A pattern you will sometimes see is a `try-finally` statement with a flag variable to indicate whether or not the body of the `finally` clause should be executed. In its simplest form (that can't already be handled just by using an `except` clause instead), it looks something like this:

```
cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()
```

As with any `try` statement based code, this can cause problems for development and review, because the setup code and the cleanup code can end up being separated by arbitrarily long sections of code.

`ExitStack` makes it possible to instead register a callback for execution at the end of a `with` statement, and then later decide to skip executing that callback:

```
from contextlib import ExitStack

with ExitStack() as stack:
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()
```

This allows the intended cleanup up behaviour to be made explicit up front, rather than requiring a separate flag variable.

If a particular application uses this pattern a lot, it can be simplified even further by means of a small helper class:

```
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, *args, **kwds):
        super(Callback, self).__init__()
        self.callback(callback, *args, **kwds)

    def cancel(self):
        self.pop_all()

with Callback(cleanup_resources) as cb:
    result = perform_operation()
    if result:
        cb.cancel()
```

If the resource cleanup isn't already neatly bundled into a standalone function, then it is still possible to use the decorator form of *`ExitStack.callback()`* to declare the resource cleanup in advance:

```
from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
    result = perform_operation()
    if result:
        stack.pop_all()
```

Due to the way the decorator protocol works, a callback function declared this way cannot take any parameters. Instead, any resources to be released must be accessed as closure variables.

Using a context manager as a function decorator

ContextDecorator makes it possible to use a context manager in both an ordinary `with` statement and also as a function decorator.

For example, it is sometimes useful to wrap functions or groups of statements with a logger that can track the time of entry and time of exit. Rather than writing both a function decorator and a context manager for the task, inheriting from *ContextDecorator* provides both capabilities in a single definition:

```
from contextlib import ContextDecorator
import logging

logging.basicConfig(level=logging.INFO)

class track_entry_and_exit(ContextDecorator):
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        logging.info('Entering: %s', self.name)

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: %s', self.name)
```

Instances of this class can be used as both a context manager:

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

And also as a function decorator:

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

Note that there is one additional limitation when using context managers as function decorators: there's no way to access the return value of `__enter__()`. If that value is needed, then it is still necessary to use an explicit `with` statement.

Ver também:

PEP 343 - The "with" statement A especificação, o histórico e os exemplos para a instrução Python `with`.

30.7.3 Single use, reusable and reentrant context managers

Most context managers are written in a way that means they can only be used effectively in a `with` statement once. These single use context managers must be created afresh each time they're used - attempting to use them a second time will trigger an exception or otherwise not work correctly.

This common limitation means that it is generally advisable to create context managers directly in the header of the `with` statement where they are used (as shown in all of the usage examples above).

Files are an example of effectively single use context managers, since the first `with` statement will close the file, preventing any further IO operations using that file object.

Context managers created using *contextmanager()* are also single use context managers, and will complain about the underlying generator failing to yield if an attempt is made to use them a second time:

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def singleuse():
...     print("Before")
...     yield
...     print("After")
...
>>> cm = singleuse()
>>> with cm:
...     pass
...
Before
After
>>> with cm:
...     pass
...
Traceback (most recent call last):
    ...
RuntimeError: generator didn't yield
```

Reentrant context managers

More sophisticated context managers may be "reentrant". These context managers can not only be used in multiple `with` statements, but may also be used *inside* a `with` statement that is already using the same context manager.

threading.RLock is an example of a reentrant context manager, as are *suppress()* and *redirect_stdout()*. Here's a very simple example of reentrant use:

```
>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> stream = StringIO()
>>> write_to_stream = redirect_stdout(stream)
>>> with write_to_stream:
...     print("This is written to the stream rather than stdout")
...     with write_to_stream:
...         print("This is also written to the stream")
...
>>> print("This is written directly to stdout")
This is written directly to stdout
>>> print(stream.getvalue())
This is written to the stream rather than stdout
This is also written to the stream
```

Real world examples of reentrancy are more likely to involve multiple functions calling each other and hence be far more complicated than this example.

Note also that being reentrant is *not* the same thing as being thread safe. *redirect_stdout()*, for example, is definitely not thread safe, as it makes a global modification to the system state by binding *sys.stdout* to a different stream.

Gerenciadores de contexto reutilizáveis

Distinct from both single use and reentrant context managers are "reusable" context managers (or, to be completely explicit, "reusable, but not reentrant" context managers, since reentrant context managers are also reusable). These context managers support being used multiple times, but will fail (or otherwise not work correctly) if the specific context manager instance has already been used in a containing with statement.

threading.Lock is an example of a reusable, but not reentrant, context manager (for a reentrant lock, it is necessary to use *threading.RLock* instead).

Another example of a reusable, but not reentrant, context manager is *ExitStack*, as it invokes *all* currently registered callbacks when leaving any with statement, regardless of where those callbacks were added:

```
>>> from contextlib import ExitStack
>>> stack = ExitStack()
>>> with stack:
...     stack.callback(print, "Callback: from first context")
...     print("Leaving first context")
...
Leaving first context
Callback: from first context
>>> with stack:
...     stack.callback(print, "Callback: from second context")
...     print("Leaving second context")
...
Leaving second context
Callback: from second context
>>> with stack:
...     stack.callback(print, "Callback: from outer context")
...     with stack:
...         stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Callback: from outer context
Leaving outer context
```

As the output from the example shows, reusing a single stack object across multiple with statements works correctly, but attempting to nest them will cause the stack to be cleared at the end of the innermost with statement, which is unlikely to be desirable behaviour.

Using separate *ExitStack* instances instead of reusing a single instance avoids that problem:

```
>>> from contextlib import ExitStack
>>> with ExitStack() as outer_stack:
...     outer_stack.callback(print, "Callback: from outer context")
...     with ExitStack() as inner_stack:
...         inner_stack.callback(print, "Callback: from inner context")
```

(continua na próxima página)

(continuação da página anterior)

```
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Leaving outer context
Callback: from outer context
```

30.8 abc — Classes Base Abstratas

Código Fonte: Lib/abc.py

Este módulo fornece a infraestrutura para definir abstract base classes (CBAs) em Python, como delineado em **PEP 3119**; veja o PEP para entender o porquê isto foi adicionado ao Python. (Veja também **PEP 3141** e o módulo *numbers* sobre uma hierarquia de tipos para números baseado nas CBAs.)

O módulo *collections* tem algumas classes concretas que derivam de CBAs; essas podem, evidentemente, ser ainda mais derivadas. Além disso, o submódulo *collections.abc* tem algumas CBAs que podem ser usadas para testar se uma classe ou instância oferece uma interface particular, por exemplo, se é hashable ou se é um mapping.

Este módulo fornece a metaclasse *ABCMeta* para definir CBAs e uma classe de ajuda *ABC* para alternativamente definir CBAs através de herança:

class abc.**ABC**

Uma classe auxiliar que tem **:classe:‘ABCMeta‘** como sua metaclasse. Com essa classe, uma classe base abstrata pode ser criada simplesmente derivando da **:classe:‘ABC‘** evitando às vezes confundir o uso da metaclasse, por exemplo:

```
from abc import ABC

class MyABC(ABC):
    pass
```

Note que o tipo de classe *ABC* ainda é **:classe:‘ABCMeta‘**, portanto herdando da **:classe:‘ABC‘** requer as precauções usuais a respeito do uso da metaclasse, como herança múltipla pode levar a conflitos de metaclasse. Pode-se também definir uma classe base abstrata ao passar a palavra reservada da metaclasse e usar **:classe:‘ABCMeta‘** diretamente, por exemplo:

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass
```

Novo na versão 3.4.

class abc.**ABCMeta**

Metaclasse para definir Classe Base Abstrata (CBAs).

Use esta metaclasse para criar uma CBA. Uma CBA pode ser diretamente subclasseada, e então agir como uma classe misturada. Você também pode registrar classes concretas não relacionadas (até mesmo classes embutidas) e CBAs não relacionadas como “subclasses virtuais” – estas e suas descendentes serão consideradas subclasses da CBA de registro pela função embutida *issubclass()*, mas a CBA de registro não irá aparecer na ORM (Ordem

de Resolução do Método) e nem as implementações do método definidas pela CBA de registro será chamável (nem mesmo via *super()*).[1]

Classes criadas com a metaclasse de **:classe:'ABCMeta'** tem o seguinte método:

register(*subclass*)
Registrar *subclasse* como uma "subclasse virtual" desta CBA. Por exemplo:

```
from abc import ABC

class MyABC(ABC):
    pass

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

Alterado na versão 3.3: Retorna a subclasse registrada, para permitir o uso como um decorador de classe.

Alterado na versão 3.4: Para detectar chamadas para *register()*, você pode usar a função *get_cache_token()*.

Você também pode sobrepor este método em uma classe base abstrata:

__subclasshook__(*subclass*)
(Deve obrigatoriamente ser definido como um método de classe.)

Cheque se a *subclasse* é considerada uma subclasse desta CBA. Isto significa que você pode customizar ainda mais o comportamento da `issubclass` sem a necessidade de chamar *register()* em toda classe que você queira considerar uma subclasse da CBA. (Este método de classe é chamado do método da CBA `__subclasscheck__()` .)

Este método deve retornar `True`, `False` ou `NotImplemented`. Se retornar `True`, a *subclasse* é considerada uma subclasse desta CBA. Se retornar `False`, a *subclasse* não é considerada uma subclasse desta CBA, mesmo que normalmente seria uma. Se retornar `NotImplemented`, a verificação da subclasse é continuada com o mecanismo usual.

Para uma demonstração destes conceitos, veja este exemplo de definição CBA:

```
class Foo:
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable(ABC):

    @abstractmethod
    def __iter__(self):
        while False:
            yield None

    def get_iterator(self):
        return self.__iter__()
```

(continua na próxima página)

[1] Programadores C++ devem notar que o conceito da casse base virtual do Python não é o mesmo que o de C++.

(continuação da página anterior)

```
    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented

MyIterable.register(Foo)
```

A CBA `MyIterable` define o método iterável padrão, *`__iter__()`*, como um método abstrato. A implementação dada aqui pode ainda ser chamada da subclasse. O método `get_iterator()` é também parte da classe base abstrata `MyIterable`, mas não precisa ser substituído nas classes derivadas não abstratas.

O método de classe *`__subclasshook__()`* definido aqui diz que qualquer classe que tenha um método `__iter__()` em seu `__dict__` (ou no de uma de suas classes base, acessados via lista `__mro__`) é considerado uma `MyIterable` também.

Finalmente, a última linha faz de `Foo` uma subclasse virtual da `MyIterable`, apesar de não definir um método `__iter__()` (ela usa o protocolo iterável antigo, definido em termos de `__len__()` e `__getitem__()`). Note que isto não fará o `get_iterator` disponível como um método de `Foo`, então ele é fornecido separadamente.

O módulo *`abc`* também fornece o seguinte decorador:

`@abc.`**`abstractmethod`**

Um decorator indicando métodos abstratos.

Usar este decorador requer que a metaclasse da classe seja **:classe:‘ABCMeta‘** ou seja derivada desta. Uma classe que tem uma metaclasse derivada de **:classe:‘ABCMeta‘** não pode ser instanciada a menos que todos os seus métodos abstratos e propriedades estejam substituídos. Os métodos abstratos podem ser chamados usando qualquer um dos mechanismos normais de ‘super’ chamadas. *`abstractmethod()`* pode ser usado para declarar métodos abstratos para propriedades e descritores.

Adicionar dinamicamente métodos abstratos a uma classe, ou tentar modificar o status de abstração de um método ou classe uma vez que estejam criados, não é tolerado. A *`abstractmethod()`* afeta apenas subclasses derivadas usando herança regular; “subclasses virtuais” registradas com o método da CBA `register()` não são afetadas.

Quando *`abstractmethod()`* é aplicado em combinação com outros descriptores de método, ele deve ser aplicado como o decorador mais interno, como mostrado nos seguintes exemplos de uso:

```
class C(ABC):
    @abstractmethod
    def my_abstract_method(self, ...):
        ...
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, ...):
        ...
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(...):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
    @my_abstract_property.setter
```

(continua na próxima página)

(continuação da página anterior)

```
    @abstractmethod
    def my_abstract_property(self, val):
        ...

    @abstractmethod
    def _get_x(self):
        ...
    @abstractmethod
    def _set_x(self, val):
        ...
    x = property(_get_x, _set_x)
```

Para que interopere corretamente com o maquinário da classe base abstrata, o descritor precisa identificar-se como abstrato usando `__isabstractmethod__`. No geral, este atributo deve ser `True` se algum dos métodos usados para compor o descritor for abstrato. Por exemplo, a **:classe:'property'** embutida do Python faz o equivalente a:

```
class Descriptor:
    ...
    @property
    def __isabstractmethod__(self):
        return any(getattr(f, '__isabstractmethod__', False) for
                   f in (self._fget, self._fset, self._fdel))
```

Nota: Diferente de métodos abstratos Java, esses métodos abstratos podem ter uma implementação. Esta implementação pode ser chamada via mecanismo da *super()* da classe que a substitui. Isto pode ser útil como um ponto final para uma super chamada em um framework que usa herança múltipla cooperativa.

O módulo *abc* também suporta os seguintes decoradores herdados:

`@abc.`**abstractclassmethod**

Novo na versão 3.2.

Obsoleto desde a versão 3.3: Agora é possível usar **:classe:'classmethod'** com *abstractmethod()*, tornando redundante este decorador.

Uma subclasse da *classmethod()* embutida, indicando um método de classe abstrato. Caso contrário, é similar à *abstractmethod()*.

Este caso especial está depreciado, pois o decorador da *classmethod()* está agora corretamente identificado como abstrato quando aplicado a um método abstrato:

```
class C(ABC):
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, ...):
        ...
```

`@abc.`**abstractstaticmethod**

Novo na versão 3.2.

Obsoleto desde a versão 3.3: Agora é possível usar **:classe:'staticmethod'** com *abstractmethod()*, tornando redundante este decorador.

Uma subclasse da *staticmethod()* embutida, indicando um método estático abstrato. Caso contrário, ela é similar à *abstractmethod()*.

Este caso especial está descontinuado, pois o decorador da *staticmethod()* está agora corretamente identificado como abstrato quando aplicado a um método abstrato:

```
class C(ABC):
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(...):
        ...
```

@abc.**abstractproperty**

Obsoleto desde a versão 3.3: Agora é possível usar **:classe:‘property‘**, `property.getter()`, `property.setter()` e `property.deleter()` com *abstractmethod()*, tornando redundante este decorador.

Uma subclasse da *property()* embutida, indicando uma propriedade abstrata.

Este caso especial está descontinuado, pois o decorador da *property()* está agora corretamente identificado como abstrato quando aplicado a um método abstrato:

```
class C(ABC):
    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
```

O exemplo acima define uma propriedade somente leitura; você também pode definir uma propriedade abstrata de leitura e gravação marcando apropriadamente um ou mais dos métodos subjacentes como abstratos:

```
class C(ABC):
    @property
    def x(self):
        ...

    @x.setter
    @abstractmethod
    def x(self, val):
        ...
```

Se apenas alguns componentes são abstratos, apenas estes componentes precisam ser atualizados para criar uma propriedade concreta em uma subclasse:

```
class D(C):
    @C.x.setter
    def x(self, val):
        ...
```

O módulo: mod: *abc* também fornece as seguintes funções:

abc.**get_cache_token**()

Retorna o token de cache da classe base abstrata atual.

O token é um objeto opaco (que suporta teste de igualdade) identificando a versão atual do cache da classe base abstrata para subclasses virtuais. O token muda a cada chamada ao *ABCMeta.register()* em qualquer CBA.

Novo na versão 3.4.

30.9 atexit — Manipuladores de Saída

O módulo *atexit* define funções para registrar e cancelar o registro de funções de limpeza. As funções assim registradas são executadas automaticamente após a conclusão normal do intérprete. O módulo *atexit* executa essas funções na ordem *reversa* na qual foram registradas; se você inscrever A, B e C, no momento do término do interpretador, eles serão executados na ordem C, B, A .

Nota: As funções registradas através deste módulo não são invocadas quando o programa é morto por um sinal não tratado pelo Python, quando um erro interno fatal do Python é detectado ou quando a função *os._exit()* é invocada.

Alterado na versão 3.7: Quando usadas com os subinterpretadores de C-API, as funções registradas são locais para o interpretador em que foram registradas.

atexit.**register**(*func*, **args*, ***kwargs*)
: Registre *func* como uma função a ser executada no término. Qualquer o argumento opcional que deve ser passado para *func* for passado como argumento para *register()*. É possível registrar mais ou menos a mesma função e argumentos.

  Na terminação normal do programa (por exemplo, se *sys.exit()* for chamado ou a execução do módulo principal for concluída), todas as funções registradas serão chamadas por último, pela primeira ordem. A suposição é que os módulos de nível inferior normalmente serão importados antes dos módulos de nível superior e, portanto, devem ser limpos posteriormente.

  Se uma exceção é levantada durante a execução dos manipuladores de saída, um traceback é impresso (a menos que *SystemExit* seja levantado) e as informações de exceção sejam salvas. Depois de todos os manipuladores de saída terem tido a chance de executar a última exceção a ser levantada, é levantada novamente.

  Esta função retorna *func*, o que torna possível usá-la como um decorador.

atexit.**unregister**(*func*)
: Remove *func* da lista de funções a serem executadas no desligamento do interpretador. Depois de chamar *unregister()*, *func* tem garantia de não ser chamado quando o interpretador é encerrado, mesmo que tenha sido registrado mais de uma vez. *unregister()* silenciosamente não faz nada se *func* não foi registrado anteriormente.

Ver também:

Módulo *readline* Exemplo útil de *atexit* para ler e escrever arquivos de histórico de *readline*.

30.9.1 Exemplo do atexit

O exemplo simples a seguir demonstra como um módulo pode inicializar um contador de um arquivo quando ele é importado e salvar automaticamente o valor atualizado do contador quando o programa termina, sem depender que a aplicação faça uma chamada explícita nesse módulo na finalização.

```
try:
    with open("counterfile") as infile:
        _count = int(infile.read())
except FileNotFoundError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n
```

(continua na próxima página)

(continuação da página anterior)

```
def savecounter():
    with open("counterfile", "w") as outfile:
        outfile.write("%d" % _count)

import atexit
atexit.register(savecounter)
```

Os argumentos posicional e de palavra reservada também podem ser passados para *register()* para ser passada para a função registrada quando é chamada

```
def goodbye(name, adjective):
    print('Goodbye, %s, it was %s to meet you.' % (name, adjective))

import atexit
atexit.register(goodbye, 'Donny', 'nice')

# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```

Utilizado como um *decorator* de função:

```
import atexit

@atexit.register
def goodbye():
    print("You are now leaving the Python sector.")
```

Isso só funciona com funções que podem ser invocadas sem argumentos.

30.10 `traceback` — Print or retrieve a stack traceback

Código Fonte: Lib/traceback.py

This module provides a standard interface to extract, format and print stack traces of Python programs. It exactly mimics the behavior of the Python interpreter when it prints a stack trace. This is useful when you want to print stack traces under program control, such as in a "wrapper" around the interpreter.

The module uses traceback objects — this is the object type that is stored in the *sys.last_traceback* variable and returned as the third item from *sys.exc_info()*.

O módulo define as seguintes funções:

`traceback.`**`print_tb`**(*tb*, *limit=None*, *file=None*)

Print up to *limit* stack trace entries from traceback object *tb* (starting from the caller's frame) if *limit* is positive. Otherwise, print the last `abs(limit)` entries. If *limit* is omitted or `None`, all entries are printed. If *file* is omitted or `None`, the output goes to `sys.stderr`; otherwise it should be an open file or file-like object to receive the output.

Alterado na versão 3.5: Added negative *limit* support.

`traceback.`**`print_exception`**(*etype*, *value*, *tb*, *limit=None*, *file=None*, *chain=True*)

Print exception information and stack trace entries from traceback object *tb* to *file*. This differs from *print_tb()* in the following ways:

- if *tb* is not `None`, it prints a header `Traceback (most recent call last):`

- it prints the exception *etype* and *value* after the stack trace
- if *type(value)* is `SyntaxError` and *value* has the appropriate format, it prints the line where the syntax error occurred with a caret indicating the approximate position of the error.

The optional *limit* argument has the same meaning as for `print_tb()`. If *chain* is true (the default), then chained exceptions (the `__cause__` or `__context__` attributes of the exception) will be printed as well, like the interpreter itself does when printing an unhandled exception.

Alterado na versão 3.5: The *etype* argument is ignored and inferred from the type of *value*.

`traceback.`**`print_exc`**(*limit=None*, *file=None*, *chain=True*)
This is a shorthand for `print_exception(*sys.exc_info(), limit, file, chain)`.

`traceback.`**`print_last`**(*limit=None*, *file=None*, *chain=True*)
This is a shorthand for `print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file, chain)`. In general it will work only after an exception has reached an interactive prompt (see `sys.last_type`).

`traceback.`**`print_stack`**(*f=None*, *limit=None*, *file=None*)
Print up to *limit* stack trace entries (starting from the invocation point) if *limit* is positive. Otherwise, print the last `abs(limit)` entries. If *limit* is omitted or `None`, all entries are printed. The optional *f* argument can be used to specify an alternate stack frame to start. The optional *file* argument has the same meaning as for `print_tb()`.

Alterado na versão 3.5: Added negative *limit* support.

`traceback.`**`extract_tb`**(*tb*, *limit=None*)
Return a `StackSummary` object representing a list of "pre-processed" stack trace entries extracted from the traceback object *tb*. It is useful for alternate formatting of stack traces. The optional *limit* argument has the same meaning as for `print_tb()`. A "pre-processed" stack trace entry is a `FrameSummary` object containing attributes `filename`, `lineno`, `name`, and `line` representing the information that is usually printed for a stack trace. The `line` is a string with leading and trailing whitespace stripped; if the source is not available it is `None`.

`traceback.`**`extract_stack`**(*f=None*, *limit=None*)
Extract the raw traceback from the current stack frame. The return value has the same format as for `extract_tb()`. The optional *f* and *limit* arguments have the same meaning as for `print_stack()`.

`traceback.`**`format_list`**(*extracted_list*)
Given a list of tuples or `FrameSummary` objects as returned by `extract_tb()` or `extract_stack()`, return a list of strings ready for printing. Each string in the resulting list corresponds to the item with the same index in the argument list. Each string ends in a newline; the strings may contain internal newlines as well, for those items whose source text line is not `None`.

`traceback.`**`format_exception_only`**(*etype*, *value*)
Format the exception part of a traceback. The arguments are the exception type and value such as given by `sys.last_type` and `sys.last_value`. The return value is a list of strings, each ending in a newline. Normally, the list contains a single string; however, for `SyntaxError` exceptions, it contains several lines that (when printed) display detailed information about where the syntax error occurred. The message indicating which exception occurred is the always last string in the list.

`traceback.`**`format_exception`**(*etype*, *value*, *tb*, *limit=None*, *chain=True*)
Format a stack trace and the exception information. The arguments have the same meaning as the corresponding arguments to `print_exception()`. The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does `print_exception()`.

Alterado na versão 3.5: The *etype* argument is ignored and inferred from the type of *value*.

`traceback.`**`format_exc`**(*limit=None*, *chain=True*)
This is like `print_exc(limit)` but returns a string instead of printing to a file.

traceback.**format_tb**(*tb*, *limit=None*)
A shorthand for `format_list(extract_tb(tb, limit))`.

traceback.**format_stack**(*f=None*, *limit=None*)
A shorthand for `format_list(extract_stack(f, limit))`.

traceback.**clear_frames**(*tb*)
Clears the local variables of all the stack frames in a traceback *tb* by calling the `clear()` method of each frame object.

Novo na versão 3.4.

traceback.**walk_stack**(*f*)
Walk a stack following `f.f_back` from the given frame, yielding the frame and line number for each frame. If *f* is `None`, the current stack is used. This helper is used with *`StackSummary.extract()`*.

Novo na versão 3.5.

traceback.**walk_tb**(*tb*)
Walk a traceback following `tb_next` yielding the frame and line number for each frame. This helper is used with *`StackSummary.extract()`*.

Novo na versão 3.5.

The module also defines the following classes:

30.10.1 `TracebackException` Objects

Novo na versão 3.5.

`TracebackException` objects are created from actual exceptions to capture data for later printing in a lightweight fashion.

class traceback.**TracebackException**(*exc_type*, *exc_value*, *exc_traceback*, ***, *limit=None*, *lookup_lines=True*, *capture_locals=False*)
Capture an exception for later rendering. *limit*, *lookup_lines* and *capture_locals* are as for the *`StackSummary`* class.

Note that when locals are captured, they are also shown in the traceback.

__cause__
A *`TracebackException`* of the original `__cause__`.

__context__
A *`TracebackException`* of the original `__context__`.

__suppress_context__
The `__suppress_context__` value from the original exception.

stack
A *`StackSummary`* representing the traceback.

exc_type
The class of the original traceback.

filename
For syntax errors - the file name where the error occurred.

lineno
For syntax errors - the line number where the error occurred.

text
For syntax errors - the text where the error occurred.

offset
For syntax errors - the offset into the text where the error occurred.

msg
For syntax errors - the compiler error message.

classmethod from_exception(*exc*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)
Capture an exception for later rendering. *limit*, *lookup_lines* and *capture_locals* are as for the *StackSummary* class.

Note that when locals are captured, they are also shown in the traceback.

format(*, *chain=True*)
Format the exception.

If *chain* is not `True`, `__cause__` and `__context__` will not be formatted.

The return value is a generator of strings, each ending in a newline and some containing internal newlines. *print_exception()* is a wrapper around this method which just prints the lines to a file.

The message indicating which exception occurred is always the last string in the output.

format_exception_only()
Format the exception part of the traceback.

The return value is a generator of strings, each ending in a newline.

Normally, the generator emits a single string; however, for *SyntaxError* exceptions, it emits several lines that (when printed) display detailed information about where the syntax error occurred.

The message indicating which exception occurred is always the last string in the output.

30.10.2 Objetos `StackSummary`

Novo na versão 3.5.

StackSummary objects represent a call stack ready for formatting.

class traceback.**StackSummary**

classmethod extract(*frame_gen*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)
Construct a *StackSummary* object from a frame generator (such as is returned by *walk_stack()* or *walk_tb()*).

If *limit* is supplied, only this many frames are taken from *frame_gen*. If *lookup_lines* is `False`, the returned *FrameSummary* objects will not have read their lines in yet, making the cost of creating the *StackSummary* cheaper (which may be valuable if it may not actually get formatted). If *capture_locals* is `True` the local variables in each *FrameSummary* are captured as object representations.

classmethod from_list(*a_list*)
Construct a *StackSummary* object from a supplied list of *FrameSummary* objects or old-style list of tuples. Each tuple should be a 4-tuple with filename, lineno, name, line as the elements.

format()
Returns a list of strings ready for printing. Each string in the resulting list corresponds to a single frame from the stack. Each string ends in a newline; the strings may contain internal newlines as well, for those items with source text lines.

For long sequences of the same frame and line, the first few repetitions are shown, followed by a summary line stating the exact number of further repetitions.

Alterado na versão 3.6: Long sequences of repeated frames are now abbreviated.

30.10.3 `FrameSummary` Objects

Novo na versão 3.5.

FrameSummary objects represent a single frame in a traceback.

class traceback.**FrameSummary**(*filename*, *lineno*, *name*, *lookup_line=True*, *locals=None*, *line=None*)

Represent a single frame in the traceback or stack that is being formatted or printed. It may optionally have a stringified version of the frames locals included in it. If *lookup_line* is `False`, the source code is not looked up until the *FrameSummary* has the `line` attribute accessed (which also happens when casting it to a tuple). `line` may be directly provided, and will prevent line lookups happening at all. *locals* is an optional local variable dictionary, and if supplied the variable representations are stored in the summary for later display.

30.10.4 Exemplos de Traceback

This simple example implements a basic read-eval-print loop, similar to (but less useful than) the standard Python interactive interpreter loop. For a more complete implementation of the interpreter loop, refer to the *code* module.

```
import sys, traceback

def run_user_code(envdir):
    source = input(">>> ")
    try:
        exec(source, envdir)
    except Exception:
        print("Exception in user code:")
        print("-"*60)
        traceback.print_exc(file=sys.stdout)
        print("-"*60)

envdir = {}
while True:
    run_user_code(envdir)
```

The following example demonstrates the different ways to print and format the exception and traceback:

```
import sys, traceback

def lumberjack():
    bright_side_of_death()

def bright_side_of_death():
    return tuple()[0]

try:
    lumberjack()
except IndexError:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print("*** print_tb:")
    traceback.print_tb(exc_traceback, limit=1, file=sys.stdout)
    print("*** print_exception:")
    # exc_type below is ignored on 3.5 and later
    traceback.print_exception(exc_type, exc_value, exc_traceback,
```

(continua na próxima página)

(continuação da página anterior)

```
                                limit=2, file=sys.stdout)
    print("*** print_exc:")
    traceback.print_exc(limit=2, file=sys.stdout)
    print("*** format_exc, first and last line:")
    formatted_lines = traceback.format_exc().splitlines()
    print(formatted_lines[0])
    print(formatted_lines[-1])
    print("*** format_exception:")
    # exc_type below is ignored on 3.5 and later
    print(repr(traceback.format_exception(exc_type, exc_value,
                                          exc_traceback)))
    print("*** extract_tb:")
    print(repr(traceback.extract_tb(exc_traceback)))
    print("*** format_tb:")
    print(repr(traceback.format_tb(exc_traceback)))
    print("*** tb_lineno:", exc_traceback.tb_lineno)
```

The output for the example would look similar to this:

```
*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** format_exc, first and last line:
Traceback (most recent call last):
IndexError: tuple index out of range
*** format_exception:
['Traceback (most recent call last):\n',
 '  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n',
 'IndexError: tuple index out of range\n']
*** extract_tb:
[<FrameSummary file <doctest...>, line 10 in <module>>,
 <FrameSummary file <doctest...>, line 4 in lumberjack>,
 <FrameSummary file <doctest...>, line 7 in bright_side_of_death>]
*** format_tb:
['  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n']
*** tb_lineno: 10
```

The following example shows the different ways to print and format the stack:

```
>>> import traceback
>>> def another_function():
...     lumberstack()
...
>>> def lumberstack():
...     traceback.print_stack()
...     print(repr(traceback.extract_stack()))
...     print(repr(traceback.format_stack()))
...
>>> another_function()
  File "<doctest>", line 10, in <module>
    another_function()
  File "<doctest>", line 3, in another_function
    lumberstack()
  File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack', 'print(repr(traceback.extract_stack()))')]
['  File "<doctest>", line 10, in <module>\n    another_function()\n',
 '  File "<doctest>", line 3, in another_function\n    lumberstack()\n',
 '  File "<doctest>", line 8, in lumberstack\n    print(repr(traceback.format_
↪stack()))\n']
```

This last example demonstrates the final few formatting functions:

```
>>> import traceback
>>> traceback.format_list([('spam.py', 3, '<module>', 'spam.eggs()'),
...                        ('eggs.py', 42, 'eggs', 'return "bacon"')])
['  File "spam.py", line 3, in <module>\n    spam.eggs()\n',
 '  File "eggs.py", line 42, in eggs\n    return "bacon"\n']
>>> an_error = IndexError('tuple index out of range')
>>> traceback.format_exception_only(type(an_error), an_error)
['IndexError: tuple index out of range\n']
```

30.11 : mod: *__future__* — Definições de declaração futura

Código-fonte: Lib/__ future __.py

: mod: *__future__* é um módulo real, e serve três propósitos:

- Para evitar confundir as ferramentas existentes que analisam as declarações de importação e esperam encontrar os módulos que estão importando.
- Para garantir que: ref: *future statements 1 'executado em versões anteriores a 2.1, pelo menos, processar exceções de tempo de execução (a importação de: mod:* __future__' falhará, porque não havia nenhum módulo desse nome antes de 2.1).
- Para documentar quando as mudanças incompatíveis foram introduzidas, e quando elas serão — ou foram — obrigatórias. Esta é uma forma de documentação executável e pode ser inspecionada programaticamente através da importação: mod: *__future__* e examinando seus conteúdos.

Cada declaração em: file:. *__future __ py* é da forma

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease,
                       CompilerFlag)
```

Onde, normalmente, * OpcionalRelease * é inferior a *MandatoryRelease*, e ambos são 5-tuplas da mesma forma que: dados: *sys.version_info*

```
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
 PY_MINOR_VERSION, # the 1; an int
 PY_MICRO_VERSION, # the 0; an int
 PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string
 PY_RELEASE_SERIAL # the 3; an int
)
```

OptionalRelease registra o primeiro lançamento no qual o recurso foi aceito.

No caso de * *MandatoryRelease* que ainda não ocorreu, *MandatoryRelease* prevê o lançamento em que o recurso se tornará parte do idioma.

Senão *MandatoryRelease* registra quando o recurso se tornou parte do idioma; Em versões em ou depois disso, os módulos não precisam mais de uma declaração futura para usar o recurso em questão, mas podem continuar a usar essas importações.

MandatoryRelease também pode ser " None", o que significa que uma característica planejada foi descartada.

Instâncias de classe: classe: *_Feature* tem dois métodos correspondentes,: meth:' getOptionalRelease' e: meth: *getMandatoryRelease*.

CompilerFlag é o sinalizador (bitfield) que deve ser passado no quarto argumento para a função incorporada: func: *compile* para habilitar o recurso no código compilado dinamicamente. Este sinalizador é armazenado em: attr: *compiler_flag* atributo em: classe:' _Feature' instâncias.

Nenhuma descrição de característica será excluída de: mod: *__future__*. Desde a sua introdução no Python 2.1, os seguintes recursos encontraram o caminho para o idioma usando esse mecanismo:

característica	Opcional em	Obrigatório em	efeito
nested_scopes	2.1.0b1	2.2	**PEP 227**: *Statically Nested Scopes*
Geradores	2.2.0a1	2.3	**PEP 255**: *Simple Generators*
divisão	2.2.0a2	3.0	**PEP 238**: *Changing the Division Operator*
absolute_import	2.5.0a1	3.0	**PEP 328**: *Imports: Multi-Line and Absolute/Relative*
with_statement	2.5.0a1	2.6	**PEP 343**: *The "with" Statement*
print_function	2.6.0a2	3.0	**PEP 3105**: *Make print a function*
unicode_literals	2.6.0a2	3.0	**PEP 3112**: *Bytes literals in Python 3000*
generator_stop	3.5.0b1	3.7	**PEP 479**: *StopIteration handling inside generators*
annotations	3.7.0b1	4.0	**PEP 563**: *Postponed evaluation of annotations*

Ver também:

future Como o compilador trata as importações futuras.

30.12 gc — Interface para o coletor de lixo

Este módulo fornece uma interface para o opcional garbage collector. Ele disponibiliza a habilidade de desabilitar o collector, ajustar a frequência da coleção, e configurar as opções de depuração. Ele também fornece acesso para objetos inacessíveis que o collector encontra mas não pode "limpar". Como o collector complementa a contagem de referência já usada em Python, você pode desabilitar o collector se você tem certeza que o seu programa não cria ciclos de referências. A coleta automática pode ser desabilitada pela chamada `gc.disable()`. Para depurar um programa vazando, chame `gc.set_debug(gc.DEBUG_LEAK)`. Perceba que isto inclui `gc.DEBUG_SAVEALL`, fazendo com que objetos coletados pelo garbage-collector sejam salvos para inspeção em gc.garbage.

O módulo *gc* fornece as seguintes funções:

`gc.`**`enable`**`()`
Habilita a coleta de lixo automática.

`gc.`**`disable`**`()`
Desative a coleta de lixo automática.

`gc.`**`isenabled`**`()`
Retorne `True` se a coleta automática estiver habilitada.

`gc.`**`collect`**`(`*generation=2*`)`
Sem argumentos, execute uma coleta completa. O argumento opcional *generation* pode ser um inteiro especificando qual geração coletar (de 0 a 2). Uma exceção *ValueError* é levantada se o número de geração for inválido. O número de objetos inacessíveis encontrados é retornado.

As listas livres mantidas para vários tipos embutidos são limpas sempre que uma coleta completa ou coleta da geração mais alta (2) é executada. Nem todos os itens em algumas listas livres podem ser liberados devido à implementação particular, em particular *float*.

`gc.`**`set_debug`**`(`*flags*`)`
Define os sinalizadores de depuração da coleta de lixo. As informações de depuração serão gravadas em `sys.stderr`. Veja abaixo uma lista de sinalizadores de depuração que podem ser combinados usando operações de bit para controlar a depuração.

`gc.`**`get_debug`**`()`
Retorne os sinalizadores de depuração atualmente definidos.

`gc.`**`get_objects`**`()`
Returns a list of all objects tracked by the collector, excluding the list returned.

`gc.`**`get_stats`**`()`
Retorna uma lista de três dicionários por geração contendo estatísticas de coleta desde o início do interpretador. O número de chaves pode mudar no futuro, mas atualmente cada dicionário conterá os seguintes itens:

- "collections"é o número de vezes que esta geração foi coletada;
- `collected` é o número total de objetos coletados nesta geração;
- `uncollectable` é o número total de objetos que foram considerados incobráveis (e, portanto, movidos para a lista *garbage*) dentro desta geração.

Novo na versão 3.4.

`gc.`**`set_threshold`**`(`*threshold0*[, *threshold1*[, *threshold2*]])
Define os limites de coleta de lixo (a frequência de coleta). Definir *threshold0* como zero desativa a coleta.

The GC classifies objects into three generations depending on how many collection sweeps they have survived. New objects are placed in the youngest generation (generation `0`). If an object survives a collection it is moved

into the next older generation. Since generation 2 is the oldest generation, objects in that generation remain there after a collection. In order to decide when to run, the collector keeps track of the number object allocations and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds *threshold0*, collection starts. Initially only generation 0 is examined. If generation 0 has been examined more than *threshold1* times since generation 1 has been examined, then generation 1 is examined as well. With the third generation, things are a bit more complicated, see Collecting the oldest generation for more information.

gc.**get_count**()
Retorna as contagens da coleta atual como uma tupla de (count0, count1, count2).

gc.**get_threshold**()
Retorne os limites da coleção atual como uma tupla de (threshold0, threshold1, threshold2).

gc.**get_referrers**(**objs*)
Retorna a lista de objetos que se referem diretamente a qualquer um dos objs. Esta função localizará apenas os contêineres que suportam coleta de lixo; tipos de extensão que se referem a outros objetos, mas não suportam coleta de lixo, não serão encontrados.

Observe que os objetos que já foram desreferenciados, mas que vivem em ciclos e ainda não foram coletados pelo coletor de lixo podem ser listados entre os referenciadores resultantes. Para obter apenas os objetos atualmente ativos, chame *collect()* antes de chamar *get_referrers()*.

Aviso: Deve-se tomar cuidado ao usar objetos retornados por *get_referrers()* porque alguns deles ainda podem estar em construção e, portanto, em um estado temporariamente inválido. Evite usar *get_referrers()* para qualquer finalidade que não seja depuração.

gc.**get_referents**(**objs*)
Retorna uma lista de objetos diretamente referenciados por qualquer um dos argumentos. Os referentes retornados são aqueles objetos visitados pelos métodos a nível do C tp_traverse dos argumentos (se houver), e podem não ser todos os objetos diretamente alcançáveis. Os métodos tp_traverse são suportados apenas por objetos que suportam coleta de lixo e são necessários apenas para visitar objetos que possam estar envolvidos em um ciclo. Assim, por exemplo, se um número inteiro pode ser acessado diretamente de um argumento, esse objeto inteiro pode ou não aparecer na lista de resultados.

gc.**is_tracked**(*obj*)
Retorna True se o objeto está atualmente rastreado pelo coletor de lixo, False caso contrário. Como regra geral, as instâncias de tipos atômicos não são rastreadas e as instâncias de tipos não atômicos (contêineres, objetos definidos pelo usuário…) são. No entanto, algumas otimizações específicas do tipo podem estar presentes para suprimir a pegada do coletor de lixo de instâncias simples (por exemplo, dicts contendo apenas chaves e valores atômicos):

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
False
>>> gc.is_tracked({"a": []})
True
```

Novo na versão 3.1.

gc.**freeze**()
Freeze all the objects tracked by gc - move them to a permanent generation and ignore all the future collections. This can be used before a POSIX fork() call to make the gc copy-on-write friendly or to speed up collection. Also collection before a POSIX fork() call may free pages for future allocation which can cause copy-on-write too so it's advised to disable gc in master process and freeze before fork and enable gc in child process.

Novo na versão 3.7.

gc.**unfreeze**()
Descongela os objetos na geração permanente, coloca-os de volta na geração mais antiga.

Novo na versão 3.7.

gc.**get_freeze_count**()
Retorna o número de objetos na geração permanente.

Novo na versão 3.7.

As seguintes variáveis são fornecidas para acesso somente leitura (você pode alterar os valores, mas não deve revinculá-los):

gc.**garbage**
Uma lista de objetos que o coletor considerou inacessíveis, mas não puderam ser liberados (objetos não coletáveis). A partir do Python 3.4, esta lista deve estar vazia na maioria das vezes, exceto ao usar instâncias de tipos de extensão C com um slot `tp_del` não-`NULL`.

Se *DEBUG_SAVEALL* for definido, todos os objetos inacessíveis serão adicionados a esta lista ao invés de liberados.

Alterado na versão 3.2: Se esta lista não estiver vazia no desligamento do interpretador, um *ResourceWarning* é emitido, que é silencioso por padrão. Se *DEBUG_UNCOLLECTABLE* for definido, além disso, todos os objetos não coletáveis serão impressos.

Alterado na versão 3.4: Seguindo a **PEP 442**, objetos com um método `__del__()` não vão mais para *gc.garbage*.

gc.**callbacks**
Uma lista de retornos de chamada que serão invocados pelo coletor de lixo antes e depois da coleta. As funções de retorno serão chamadas com dois argumentos, *phase* e *info*.

phase pode ser um dos dois valores:

"start": A coleta de lixo está prestes a começar.

"stop": A coleta de lixo terminou.

info é um ditado que fornece mais informações para a função de retorno. As seguintes chaves estão atualmente definidas:

"generation": A geração mais antiga sendo coletada.

"collected": Quando *phase* é "stop", o número de objetos coletados com sucesso.

"uncollectable": Quando *phase* é "stop", o número de objetos que não puderam ser coletados e foram colocados em *garbage*.

As aplicações podem adicionar suas próprias funções de retorno a essa lista. Os principais casos de uso são:

Reunir estatísticas sobre coleta de lixo, como com que frequência várias gerações são coletadas e quanto tempo leva a coleta.

Permitindo que os aplicativos identifiquem e limpem seus próprios tipos não colecionáveis quando eles aparecem em *garbage*.

Novo na versão 3.3.

As seguintes constantes são fornecidas para uso com *set_debug()*:

gc.**DEBUG_STATS**
Imprimir estatísticas durante a coleta. Esta informação pode ser útil ao sintonizar a frequência de coleta.

gc.**DEBUG_COLLECTABLE**
Imprimir informações sobre objetos colecionáveis encontrados.

gc.**DEBUG_UNCOLLECTABLE**
Imprime informações de objetos não colecionáveis encontrados (objetos que não são alcançáveis, mas não podem ser liberados pelo coletor). Esses objetos serão adicionados à lista `garbage`.

Alterado na versão 3.2: Imprime também o conteúdo da lista *garbage* em desligamento do interpretador, se não estiver vazia.

gc.**DEBUG_SAVEALL**
Quando definido, todos os objetos inacessíveis encontrados serão anexados ao *lixo* em vez de serem liberados. Isso pode ser útil para depurar um programa com vazamento.

gc.**DEBUG_LEAK**
Os sinalizadores de depuração necessários para o coletor imprimir informações sobre um programa com vazamento (igual a `DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_SAVEALL`).

30.13 `inspect` — Inspeciona objetos vivos

Código Fonte: Lib/inspect.py

The *inspect* module provides several useful functions to help get information about live objects such as modules, classes, methods, functions, tracebacks, frame objects, and code objects. For example, it can help you examine the contents of a class, retrieve the source code of a method, extract and format the argument list for a function, or get all the information you need to display a detailed traceback.

There are four main kinds of services provided by this module: type checking, getting source code, inspecting classes and functions, and examining the interpreter stack.

30.13.1 Tipos e membros

The *getmembers()* function retrieves the members of an object such as a class or module. The functions whose names begin with "is" are mainly provided as convenient choices for the second argument to *getmembers()*. They also help you determine when you can expect to find the following special attributes:

Tipo	Atributo	Description (descrição)
módulo	__doc__	string de documentação
	__file__	nome de arquivo (faltando para módulos embutidos)
Classe	__doc__	string de documentação
	__name__	nome com o qual esta classe foi definida
	__qualname__	qualified name (nome qualificado)
	__module__	nome do módulo no qual esta classe foi definida
method (método)	__doc__	string de documentação
	__name__	nome com o qual este método foi definido
	__qualname__	qualified name (nome qualificado)
	__func__	objeto função contendo implementação de método

Continuação na pró

Tabela 1 – continuação da página anterior

Tipo	Atributo	Description (descrição)
	__self__	instância para o qual este método está vinculado, ou `None`.
	__module__	nome do módulo no qual este método foi definido
function (função)	__doc__	string de documentação
	__name__	nome com o qual esta função foi definida
	__qualname__	qualified name (nome qualificado)
	__code__	code object containing compiled function *bytecode*
	__defaults__	tuple of any default values for positional or keyword parameters
	__kwdefaults__	mapping of any default values for keyword-only parameters
	__globals__	global namespace in which this function was defined
	__annotations__	mapping of parameters names to annotations; `"return"` key is reserved for return annotatio
	__module__	name of module in which this function was defined
traceback	tb_frame	frame object at this level
	tb_lasti	index of last attempted instruction in bytecode
	tb_lineno	current line number in Python source code
	tb_next	next inner traceback object (called by this level)
frame	f_back	next outer frame object (this frame's caller)
	f_builtins	builtins namespace seen by this frame
	f_code	code object being executed in this frame
	f_globals	global namespace seen by this frame
	f_lasti	index of last attempted instruction in bytecode
	f_lineno	current line number in Python source code
	f_locals	local namespace seen by this frame
	f_trace	tracing function for this frame, or `None`
code	co_argcount	number of arguments (not including keyword only arguments, * or ** args)
	co_code	string of raw compiled bytecode
	co_cellvars	tuple of names of cell variables (referenced by containing scopes)
	co_consts	tuple of constants used in the bytecode
	co_filename	name of file in which this code object was created
	co_firstlineno	number of first line in Python source code
	co_flags	bitmap of `CO_*` flags, read more *here*
	co_lnotab	encoded mapping of line numbers to bytecode indices
	co_freevars	tuple of names of free variables (referenced via a function's closure)
	co_kwonlyargcount	number of keyword only arguments (not including ** arg)
	co_name	name with which this code object was defined
	co_names	tuple of names of local variables
	co_nlocals	number of local variables
	co_stacksize	virtual machine stack space required
	co_varnames	tuple of names of arguments and local variables
gerador	__name__	nome
	__qualname__	qualified name (nome qualificado)
	gi_frame	frame
	gi_running	is the generator running?
	gi_code	code
	gi_yieldfrom	object being iterated by `yield from`, or `None`
co-rotina	__name__	nome
	__qualname__	qualified name (nome qualificado)
	cr_await	object being awaited on, or `None`
	cr_frame	frame
	cr_running	is the coroutine running?

Continuação na pró

Tabela 1 – continuação da página anterior

Tipo	Atributo	Description (descrição)
	cr_code	code
	cr_origin	where coroutine was created, or `None`. See *sys.set_coroutine_origin_trackin*
builtin	__doc__	string de documentação
	__name__	original name of this function or method
	__qualname__	qualified name (nome qualificado)
	__self__	instance to which a method is bound, or `None`

Alterado na versão 3.5: Add `__qualname__` and `gi_yieldfrom` attributes to generators.

The `__name__` attribute of generators is now set from the function name, instead of the code name, and it can now be modified.

Alterado na versão 3.7: Add `cr_origin` attribute to coroutines.

`inspect.`**`getmembers`**(*object*[, *predicate*])

Return all the members of an object in a list of (name, value) pairs sorted by name. If the optional *predicate* argument is supplied, only members for which the predicate returns a true value are included.

Nota: *getmembers()* will only return class attributes defined in the metaclass when the argument is a class and those attributes have been listed in the metaclass' custom `__dir__()`.

`inspect.`**`getmodulename`**(*path*)

Return the name of the module named by the file *path*, without including the names of enclosing packages. The file extension is checked against all of the entries in *importlib.machinery.all_suffixes()*. If it matches, the final path component is returned with the extension removed. Otherwise, `None` is returned.

Note that this function *only* returns a meaningful name for actual Python modules - paths that potentially refer to Python packages will still return `None`.

Alterado na versão 3.3: The function is based directly on *importlib*.

`inspect.`**`ismodule`**(*object*)

Return `True` if the object is a module.

`inspect.`**`isclass`**(*object*)

Return `True` if the object is a class, whether built-in or created in Python code.

`inspect.`**`ismethod`**(*object*)

Return `True` if the object is a bound method written in Python.

`inspect.`**`isfunction`**(*object*)

Return `True` if the object is a Python function, which includes functions created by a *lambda* expression.

`inspect.`**`isgeneratorfunction`**(*object*)

Return `True` if the object is a Python generator function.

`inspect.`**`isgenerator`**(*object*)

Return `True` if the object is a generator.

`inspect.`**`iscoroutinefunction`**(*object*)

Return `True` if the object is a *coroutine function* (a function defined with an `async def` syntax).

Novo na versão 3.5.

`inspect.`**`iscoroutine`**(*object*)

Return `True` if the object is a *coroutine* created by an `async def` function.

Novo na versão 3.5.

inspect.**isawaitable**(*object*)
Return `True` if the object can be used in `await` expression.

Can also be used to distinguish generator-based coroutines from regular generators:

```
def gen():
    yield
@types.coroutine
def gen_coro():
    yield

assert not isawaitable(gen())
assert isawaitable(gen_coro())
```

Novo na versão 3.5.

inspect.**isasyncgenfunction**(*object*)
Return `True` if the object is an *asynchronous generator* function, for example:

```
>>> async def agen():
...     yield 1
...
>>> inspect.isasyncgenfunction(agen)
True
```

Novo na versão 3.6.

inspect.**isasyncgen**(*object*)
Return `True` if the object is an *asynchronous generator iterator* created by an *asynchronous generator* function.

Novo na versão 3.6.

inspect.**istraceback**(*object*)
Return `True` if the object is a traceback.

inspect.**isframe**(*object*)
Return `True` if the object is a frame.

inspect.**iscode**(*object*)
Return `True` if the object is a code.

inspect.**isbuiltin**(*object*)
Return `True` if the object is a built-in function or a bound built-in method.

inspect.**isroutine**(*object*)
Return `True` if the object is a user-defined or built-in function or method.

inspect.**isabstract**(*object*)
Return `True` if the object is an abstract base class.

inspect.**ismethoddescriptor**(*object*)
Return `True` if the object is a method descriptor, but not if *ismethod()*, *isclass()*, *isfunction()* or *isbuiltin()* are true.

This, for example, is true of `int.__add__`. An object passing this test has a `__get__()` method but not a `__set__()` method, but beyond that the set of attributes varies. A *__name__* attribute is usually sensible, and `__doc__` often is.

Methods implemented via descriptors that also pass one of the other tests return `False` from the *ismethoddescriptor()* test, simply because the other tests promise more – you can, e.g., count on having the `__func__` attribute (etc) when an object passes *ismethod()*.

inspect.**isdatadescriptor**(*object*)
Return `True` if the object is a data descriptor.

Data descriptors have both a `__get__` and a `__set__` method. Examples are properties (defined in Python), getsets, and members. The latter two are defined in C and there are more specific tests available for those types, which is robust across Python implementations. Typically, data descriptors will also have *`__name__`* and `__doc__` attributes (properties, getsets, and members have both of these attributes), but this is not guaranteed.

inspect.**isgetsetdescriptor**(*object*)
Return `True` if the object is a getset descriptor.

CPython implementation detail: getsets are attributes defined in extension modules via `PyGetSetDef` structures. For Python implementations without such types, this method will always return `False`.

inspect.**ismemberdescriptor**(*object*)
Return `True` if the object is a member descriptor.

CPython implementation detail: Member descriptors are attributes defined in extension modules via `PyMemberDef` structures. For Python implementations without such types, this method will always return `False`.

30.13.2 Retrieving source code

inspect.**getdoc**(*object*)
Get the documentation string for an object, cleaned up with *`cleandoc()`*. If the documentation string for an object is not provided and the object is a class, a method, a property or a descriptor, retrieve the documentation string from the inheritance hierarchy.

Alterado na versão 3.5: Strings de documentação agora são herdadas, se não forem sobrescritas.

inspect.**getcomments**(*object*)
Return in a single string any lines of comments immediately preceding the object's source code (for a class, function, or method), or at the top of the Python source file (if the object is a module). If the object's source code is unavailable, return `None`. This could happen if the object has been defined in C or the interactive shell.

inspect.**getfile**(*object*)
Retorna o nome do arquivo (texto ou binário) no qual um objeto foi definido. Isso falhará com um: exc: *TypeError* se o objeto for um módulo, classe ou função embutidos.

inspect.**getmodule**(*object*)
Try to guess which module an object was defined in.

inspect.**getsourcefile**(*object*)
Return the name of the Python source file in which an object was defined. This will fail with a *`TypeError`* if the object is a built-in module, class, or function.

inspect.**getsourcelines**(*object*)
Return a list of source lines and starting line number for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An *`OSError`* is raised if the source code cannot be retrieved.

Alterado na versão 3.3: *`OSError`* is raised instead of *`IOError`*, now an alias of the former.

inspect.**getsource**(*object*)
Return the text of the source code for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An *`OSError`* is raised if the source code cannot be retrieved.

Alterado na versão 3.3: *`OSError`* is raised instead of *`IOError`*, now an alias of the former.

inspect.**cleandoc**(*doc*)
Clean up indentation from docstrings that are indented to line up with blocks of code.

All leading whitespace is removed from the first line. Any leading whitespace that can be uniformly removed from the second line onwards is removed. Empty lines at the beginning and end are subsequently removed. Also, all tabs are expanded to spaces.

30.13.3 Introspecting callables with the Signature object

Novo na versão 3.3.

The Signature object represents the call signature of a callable object and its return annotation. To retrieve a Signature object, use the *signature()* function.

inspect.**signature**(*callable*, *, *follow_wrapped=True*)
Return a *Signature* object for the given `callable`:

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass

>>> sig = signature(foo)

>>> str(sig)
'(a, *, b:int, **kwargs)'

>>> str(sig.parameters['b'])
'b:int'

>>> sig.parameters['b'].annotation
<class 'int'>
```

Accepts a wide range of Python callables, from plain functions and classes to *functools.partial()* objects.

Raises *ValueError* if no signature can be provided, and *TypeError* if that type of object is not supported.

A slash(/) in the signature of a function denotes that the parameters prior to it are positional-only. For more info, see the FAQ entry on positional-only parameters.

Novo na versão 3.5: `follow_wrapped` parameter. Pass `False` to get a signature of `callable` specifically (`callable.__wrapped__` will not be used to unwrap decorated callables.)

Nota: Some callables may not be introspectable in certain implementations of Python. For example, in CPython, some built-in functions defined in C provide no metadata about their arguments.

---

class inspect.**Signature**(*parameters=None*, *, *return_annotation=Signature.empty*)
A Signature object represents the call signature of a function and its return annotation. For each parameter accepted by the function it stores a *Parameter* object in its *parameters* collection.

The optional *parameters* argument is a sequence of *Parameter* objects, which is validated to check that there are no parameters with duplicate names, and that the parameters are in the right order, i.e. positional-only first, then positional-or-keyword, and that parameters with defaults follow parameters without defaults.

The optional *return_annotation* argument, can be an arbitrary Python object, is the "return" annotation of the callable.

Signature objects are *immutable*. Use *Signature.replace()* to make a modified copy.

Alterado na versão 3.5: Signature objects are picklable and hashable.

empty
A special class-level marker to specify absence of a return annotation.

parameters
An ordered mapping of parameters' names to the corresponding *Parameter* objects. Parameters appear in strict definition order, including keyword-only parameters.

Alterado na versão 3.7: Python only explicitly guaranteed that it preserved the declaration order of keyword-only parameters as of version 3.7, although in practice this order had always been preserved in Python 3.

return_annotation
The "return" annotation for the callable. If the callable has no "return" annotation, this attribute is set to *Signature.empty*.

bind(**args*, ***kwargs*)
Create a mapping from positional and keyword arguments to parameters. Returns *BoundArguments* if `*args` and `**kwargs` match the signature, or raises a *TypeError*.

bind_partial(**args*, ***kwargs*)
Works the same way as *Signature.bind()*, but allows the omission of some required arguments (mimics *functools.partial()* behavior.) Returns *BoundArguments*, or raises a *TypeError* if the passed arguments do not match the signature.

replace(**[, parameters][, return_annotation]*)
Create a new Signature instance based on the instance replace was invoked on. It is possible to pass different `parameters` and/or `return_annotation` to override the corresponding properties of the base signature. To remove return_annotation from the copied Signature, pass in *Signature.empty*.

```
>>> def test(a, b):
...     pass
>>> sig = signature(test)
>>> new_sig = sig.replace(return_annotation="new return anno")
>>> str(new_sig)
"(a, b) -> 'new return anno'"
```

classmethod from_callable(*obj*, *, *follow_wrapped=True*)
Return a *Signature* (or its subclass) object for a given callable `obj`. Pass `follow_wrapped=False` to get a signature of `obj` without unwrapping its `__wrapped__` chain.

This method simplifies subclassing of *Signature*:

```
class MySignature(Signature):
    pass
sig = MySignature.from_callable(min)
assert isinstance(sig, MySignature)
```

Novo na versão 3.5.

class inspect.**Parameter**(*name*, *kind*, *, *default=Parameter.empty*, *annotation=Parameter.empty*)
Parameter objects are *immutable*. Instead of modifying a Parameter object, you can use *Parameter.replace()* to create a modified copy.

Alterado na versão 3.5: Parameter objects are picklable and hashable.

empty
A special class-level marker to specify absence of default values and annotations.

name
The name of the parameter as a string. The name must be a valid Python identifier.

CPython implementation detail: CPython generates implicit parameter names of the form `.0` on the code objects used to implement comprehensions and generator expressions.

Alterado na versão 3.6: These parameter names are exposed by this module as names like `implicit0`.

default
The default value for the parameter. If the parameter has no default value, this attribute is set to *Parameter.empty*.

annotation
The annotation for the parameter. If the parameter has no annotation, this attribute is set to *Parameter.empty*.

kind
Describes how argument values are bound to the parameter. Possible values (accessible via *Parameter*, like `Parameter.KEYWORD_ONLY`):

Nome	Significado
POSITIONAL_ONLY	Value must be supplied as a positional argument. Python has no explicit syntax for defining positional-only parameters, but many built-in and extension module functions (especially those that accept only one or two parameters) accept them.
POSITIONAL_OR_KEYWORD	Value may be supplied as either a keyword or positional argument (this is the standard binding behaviour for functions implemented in Python.)
`` `*VAR_POSITIONAL*` ``	A tuple of positional arguments that aren't bound to any other parameter. This corresponds to a `*args` parameter in a Python function definition.
KEYWORD_ONLY	Value must be supplied as a keyword argument. Keyword only parameters are those which appear after a `*` or `*args` entry in a Python function definition.
VAR_KEYWORD	A dict of keyword arguments that aren't bound to any other parameter. This corresponds to a `**kwargs` parameter in a Python function definition.

Example: print all keyword-only arguments without default values:

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     if (param.kind == param.KEYWORD_ONLY and
...                       param.default is param.empty):
...         print('Parameter:', param)
Parameter: c
```

replace(**[, name][, kind][, default][, annotation]*)

Create a new Parameter instance based on the instance replaced was invoked on. To override a *Parameter* attribute, pass the corresponding argument. To remove a default value or/and an annotation from a Parameter, pass *Parameter.empty*.

```
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY, default=42)
>>> str(param)
```

(continua na próxima página)

(continuação da página anterior)

```
'foo=42'

>>> str(param.replace()) # Will create a shallow copy of 'param'
'foo=42'

>>> str(param.replace(default=Parameter.empty, annotation='spam'))
"foo:'spam'"
```

Alterado na versão 3.4: In Python 3.3 Parameter objects were allowed to have `name` set to `None` if their `kind` was set to `POSITIONAL_ONLY`. This is no longer permitted.

class inspect.**BoundArguments**

Result of a *Signature.bind()* or *Signature.bind_partial()* call. Holds the mapping of arguments to the function's parameters.

arguments

An ordered, mutable mapping (*collections.OrderedDict*) of parameters' names to arguments' values. Contains only explicitly bound arguments. Changes in *arguments* will reflect in *args* and *kwargs*.

Should be used in conjunction with *Signature.parameters* for any argument processing purposes.

Nota: Arguments for which *Signature.bind()* or *Signature.bind_partial()* relied on a default value are skipped. However, if needed, use *BoundArguments.apply_defaults()* to add them.

---

args

A tuple of positional arguments values. Dynamically computed from the *arguments* attribute.

kwargs

A dict of keyword arguments values. Dynamically computed from the *arguments* attribute.

signature

A reference to the parent *Signature* object.

apply_defaults()

Set default values for missing arguments.

For variable-positional arguments (`*args`) the default is an empty tuple.

For variable-keyword arguments (`**kwargs`) the default is an empty dict.

```
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
>>> ba.arguments
OrderedDict([('a', 'spam'), ('b', 'ham'), ('args', ())])
```

Novo na versão 3.5.

The *args* and *kwargs* properties can be used to invoke functions:

```
def test(a, *, b):
    ...

sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **ba.kwargs)
```

Ver também:

PEP 362 - Function Signature Object. The detailed specification, implementation details and examples.

30.13.4 Classes e funções

`inspect.`**`getclasstree`**(*classes*, *unique=False*)

Arrange the given list of classes into a hierarchy of nested lists. Where a nested list appears, it contains classes derived from the class whose entry immediately precedes the list. Each entry is a 2-tuple containing a class and a tuple of its base classes. If the *unique* argument is true, exactly one entry appears in the returned structure for each class in the given list. Otherwise, classes using multiple inheritance and their descendants will appear multiple times.

`inspect.`**`getargspec`**(*func*)

Get the names and default values of a Python function's parameters. A *named tuple* `ArgSpec(args, varargs, keywords, defaults)` is returned. *args* is a list of the parameter names. *varargs* and *keywords* are the names of the `*` and `**` parameters or `None`. *defaults* is a tuple of default argument values or `None` if there are no default arguments; if this tuple has *n* elements, they correspond to the last *n* elements listed in *args*.

Obsoleto desde a versão 3.0: Use *`getfullargspec()`* for an updated API that is usually a drop-in replacement, but also correctly handles function annotations and keyword-only parameters.

Alternatively, use *`signature()`* and *Signature Object*, which provide a more structured introspection API for callables.

`inspect.`**`getfullargspec`**(*func*)

Get the names and default values of a Python function's parameters. A *named tuple* is returned:

```
FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults,
annotations)
```

args is a list of the positional parameter names. *varargs* is the name of the `*` parameter or `None` if arbitrary positional arguments are not accepted. *varkw* is the name of the `**` parameter or `None` if arbitrary keyword arguments are not accepted. *defaults* is an *n*-tuple of default argument values corresponding to the last *n* positional parameters, or `None` if there are no such defaults defined. *kwonlyargs* is a list of keyword-only parameter names in declaration order. *kwonlydefaults* is a dictionary mapping parameter names from *kwonlyargs* to the default values used if no argument is supplied. *annotations* is a dictionary mapping parameter names to annotations. The special key `"return"` is used to report the function return value annotation (if any).

Note that *`signature()`* and *Signature Object* provide the recommended API for callable introspection, and support additional behaviours (like positional-only arguments) that are sometimes encountered in extension module APIs. This function is retained primarily for use in code that needs to maintain compatibility with the Python 2 `inspect` module API.

Alterado na versão 3.4: This function is now based on *`signature()`*, but still ignores `__wrapped__` attributes and includes the already bound first parameter in the signature output for bound methods.

Alterado na versão 3.6: This method was previously documented as deprecated in favour of *`signature()`* in Python 3.5, but that decision has been reversed in order to restore a clearly supported standard interface for single-source Python 2/3 code migrating away from the legacy *`getargspec()`* API.

Alterado na versão 3.7: Python only explicitly guaranteed that it preserved the declaration order of keyword-only parameters as of version 3.7, although in practice this order had always been preserved in Python 3.

`inspect.`**`getargvalues`**(*frame*)

Get information about arguments passed into a particular frame. A *named tuple* `ArgInfo(args, varargs, keywords, locals)` is returned. *args* is a list of the argument names. *varargs* and *keywords* are the names of the `*` and `**` arguments or `None`. *locals* is the locals dictionary of the given frame.

Nota: This function was inadvertently marked as deprecated in Python 3.5.

inspect.**formatargspec**(*args*[, *varargs*, *varkw*, *defaults*, *kwonlyargs*, *kwonlydefaults*, *annotations*[, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*, *formatreturns*, *formatannotations*]])

Format a pretty argument spec from the values returned by `getfullargspec()`.

The first seven arguments are (`args`, `varargs`, `varkw`, `defaults`, `kwonlyargs`, `kwonlydefaults`, `annotations`).

The other six arguments are functions that are called to turn argument names, `*` argument name, `**` argument name, default values, return annotation and individual annotations into strings, respectively.

Por exemplo:

```
>>> from inspect import formatargspec, getfullargspec
>>> def f(a: int, b: float):
...     pass
...
>>> formatargspec(*getfullargspec(f))
'(a: int, b: float)'
```

Obsoleto desde a versão 3.5: Use `signature()` and *Signature Object*, which provide a better introspecting API for callables.

inspect.**formatargvalues**(*args*[, *varargs*, *varkw*, *locals*, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*])

Format a pretty argument spec from the four values returned by `getargvalues()`. The format* arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

Nota: This function was inadvertently marked as deprecated in Python 3.5.

inspect.**getmro**(*cls*)

Return a tuple of class cls's base classes, including cls, in method resolution order. No class appears more than once in this tuple. Note that the method resolution order depends on cls's type. Unless a very peculiar user-defined metatype is in use, cls will be the first element of the tuple.

inspect.**getcallargs**(*func*, **args*, ***kwds*)

Bind the *args* and *kwds* to the argument names of the Python function or method *func*, as if it was called with them. For bound methods, bind also the first argument (typically named `self`) to the associated instance. A dict is returned, mapping the argument names (including the names of the `*` and `**` arguments, if any) to their values from *args* and *kwds*. In case of invoking *func* incorrectly, i.e. whenever `func(*args, **kwds)` would raise an exception because of incompatible signature, an exception of the same type and the same or similar message is raised. For example:

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
>>> getcallargs(f, 1, 2, 3) == {'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
True
>>> getcallargs(f, a=2, x=4) == {'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
True
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() missing 1 required positional argument: 'a'
```

Novo na versão 3.2.

Obsoleto desde a versão 3.5: Use *Signature.bind()* and *Signature.bind_partial()* instead.

inspect.**getclosurevars**(*func*)
Get the mapping of external name references in a Python function or method *func* to their current values. A *named tuple* `ClosureVars(nonlocals, globals, builtins, unbound)` is returned. *nonlocals* maps referenced names to lexical closure variables, *globals* to the function's module globals and *builtins* to the builtins visible from the function body. *unbound* is the set of names referenced in the function that could not be resolved at all given the current module globals and builtins.

TypeError is raised if *func* is not a Python function or method.

Novo na versão 3.3.

inspect.**unwrap**(*func*, *, *stop=None*)
Get the object wrapped by *func*. It follows the chain of `__wrapped__` attributes returning the last object in the chain.

stop is an optional callback accepting an object in the wrapper chain as its sole argument that allows the unwrapping to be terminated early if the callback returns a true value. If the callback never returns a true value, the last object in the chain is returned as usual. For example, *signature()* uses this to stop unwrapping if any object in the chain has a `__signature__` attribute defined.

ValueError is raised if a cycle is encountered.

Novo na versão 3.4.

30.13.5 A pilha to interpretador

When the following functions return "frame records," each record is a *named tuple* `FrameInfo(frame, filename, lineno, function, code_context, index)`. The tuple contains the frame object, the filename, the line number of the current line, the function name, a list of lines of context from the source code, and the index of the current line within that list.

Alterado na versão 3.5: Return a named tuple instead of a tuple.

Nota: Keeping references to frame objects, as found in the first element of the frame records these functions return, can cause your program to create reference cycles. Once a reference cycle has been created, the lifespan of all objects which can be accessed from the objects which form the cycle can become much longer even if Python's optional cycle detector is enabled. If such cycles must be created, it is important to ensure they are explicitly broken to avoid the delayed destruction of objects and increased memory consumption which occurs.

Though the cycle detector will catch these, destruction of the frames (and local variables) can be made deterministic by removing the cycle in a `finally` clause. This is also important if the cycle detector was disabled when Python was compiled or using *gc.disable()*. For example:

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

If you want to keep the frame around (for example to print a traceback later), you can also break reference cycles by using the `frame.clear()` method.

The optional *context* argument supported by most of these functions specifies the number of lines of context to return, which are centered around the current line.

`inspect.`**`getframeinfo`**(*frame*, *context=1*)
Get information about a frame or traceback object. A *named tuple* `Traceback(filename, lineno, function, code_context, index)` is returned.

`inspect.`**`getouterframes`**(*frame*, *context=1*)
Get a list of frame records for a frame and all outer frames. These frames represent the calls that lead to the creation of *frame*. The first entry in the returned list represents *frame*; the last entry represents the outermost call on *frame*'s stack.

Alterado na versão 3.5: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

`inspect.`**`getinnerframes`**(*traceback*, *context=1*)
Get a list of frame records for a traceback's frame and all inner frames. These frames represent calls made as a consequence of *frame*. The first entry in the list represents *traceback*; the last entry represents where the exception was raised.

Alterado na versão 3.5: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

`inspect.`**`currentframe`**()
Return the frame object for the caller's stack frame.

CPython implementation detail: This function relies on Python stack frame support in the interpreter, which isn't guaranteed to exist in all implementations of Python. If running in an implementation without Python stack frame support this function returns `None`.

`inspect.`**`stack`**(*context=1*)
Return a list of frame records for the caller's stack. The first entry in the returned list represents the caller; the last entry represents the outermost call on the stack.

Alterado na versão 3.5: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

`inspect.`**`trace`**(*context=1*)
Return a list of frame records for the stack between the current frame and the frame in which an exception currently being handled was raised in. The first entry in the list represents the caller; the last entry represents where the exception was raised.

Alterado na versão 3.5: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

30.13.6 Fetching attributes statically

Both *`getattr()`* and *`hasattr()`* can trigger code execution when fetching or checking for the existence of attributes. Descriptors, like properties, will be invoked and `__getattr__()` and `__getattribute__()` may be called.

For cases where you want passive introspection, like documentation tools, this can be inconvenient. *`getattr_static()`* has the same signature as *`getattr()`* but avoids executing code when it fetches attributes.

`inspect.`**`getattr_static`**(*obj*, *attr*, *default=None*)
Retrieve attributes without triggering dynamic lookup via the descriptor protocol, `__getattr__()` or `__getattribute__()`.

Note: this function may not be able to retrieve all attributes that getattr can fetch (like dynamically created attributes) and may find attributes that getattr can't (like descriptors that raise AttributeError). It can also return descriptors objects instead of instance members.

If the instance *__dict__* is shadowed by another member (for example a property) then this function will be unable to find instance members.

Novo na versão 3.2.

getattr_static() does not resolve descriptors, for example slot descriptors or getset descriptors on objects implemented in C. The descriptor object is returned instead of the underlying attribute.

You can handle these with code like the following. Note that for arbitrary getset descriptors invoking these may trigger code execution:

```
# example code for resolving the builtin descriptor types
class _foo:
    __slots__ = ['foo']

slot_descriptor = type(_foo.foo)
getset_descriptor = type(type(open(__file__)).name)
wrapper_descriptor = type(str.__dict__['__add__'])
descriptor_types = (slot_descriptor, getset_descriptor, wrapper_descriptor)

result = getattr_static(some_object, 'foo')
if type(result) in descriptor_types:
    try:
        result = result.__get__()
    except AttributeError:
        # descriptors can raise AttributeError to
        # indicate there is no underlying value
        # in which case the descriptor itself will
        # have to do
        pass
```

30.13.7 Current State of Generators and Coroutines

When implementing coroutine schedulers and for other advanced uses of generators, it is useful to determine whether a generator is currently executing, is waiting to start or resume or execution, or has already terminated. *getgeneratorstate()* allows the current state of a generator to be determined easily.

`inspect.`**`getgeneratorstate`**(*generator*)

Get current state of a generator-iterator.

Possible states are:

- GEN_CREATED: Waiting to start execution.
- GEN_RUNNING: Currently being executed by the interpreter.
- GEN_SUSPENDED: Currently suspended at a yield expression.
- GEN_CLOSED: Execution has completed.

Novo na versão 3.2.

`inspect.`**`getcoroutinestate`**(*coroutine*)

Get current state of a coroutine object. The function is intended to be used with coroutine objects created by `async def` functions, but will accept any coroutine-like object that has `cr_running` and `cr_frame` attributes.

Possible states are:

- CORO_CREATED: Waiting to start execution.
- CORO_RUNNING: Currently being executed by the interpreter.
- CORO_SUSPENDED: Currently suspended at an await expression.
- CORO_CLOSED: Execution has completed.

Novo na versão 3.5.

The current internal state of the generator can also be queried. This is mostly useful for testing purposes, to ensure that internal state is being updated as expected:

inspect.**getgeneratorlocals**(*generator*)
Get the mapping of live local variables in *generator* to their current values. A dictionary is returned that maps from variable names to values. This is the equivalent of calling *locals()* in the body of the generator, and all the same caveats apply.

If *generator* is a *generator* with no currently associated frame, then an empty dictionary is returned. *TypeError* is raised if *generator* is not a Python generator object.

CPython implementation detail: This function relies on the generator exposing a Python stack frame for introspection, which isn't guaranteed to be the case in all implementations of Python. In such cases, this function will always return an empty dictionary.

Novo na versão 3.3.

inspect.**getcoroutinelocals**(*coroutine*)
This function is analogous to *getgeneratorlocals()*, but works for coroutine objects created by `async def` functions.

Novo na versão 3.5.

30.13.8 Code Objects Bit Flags

Python code objects have a `co_flags` attribute, which is a bitmap of the following flags:

inspect.**CO_OPTIMIZED**
The code object is optimized, using fast locals.

inspect.**CO_NEWLOCALS**
If set, a new dict will be created for the frame's `f_locals` when the code object is executed.

inspect.**CO_VARARGS**
The code object has a variable positional parameter (`*args`-like).

inspect.**CO_VARKEYWORDS**
The code object has a variable keyword parameter (`**kwargs`-like).

inspect.**CO_NESTED**
The flag is set when the code object is a nested function.

inspect.**CO_GENERATOR**
The flag is set when the code object is a generator function, i.e. a generator object is returned when the code object is executed.

inspect.**CO_NOFREE**
The flag is set if there are no free or cell variables.

inspect.**CO_COROUTINE**
The flag is set when the code object is a coroutine function. When the code object is executed it returns a coroutine object. See **PEP 492** for more details.

Novo na versão 3.5.

`inspect.`**`CO_ITERABLE_COROUTINE`**
The flag is used to transform generators into generator-based coroutines. Generator objects with this flag can be used in `await` expression, and can `yield from` coroutine objects. See **PEP 492** for more details.

Novo na versão 3.5.

`inspect.`**`CO_ASYNC_GENERATOR`**
The flag is set when the code object is an asynchronous generator function. When the code object is executed it returns an asynchronous generator object. See **PEP 525** for more details.

Novo na versão 3.6.

Nota: The flags are specific to CPython, and may not be defined in other Python implementations. Furthermore, the flags are an implementation detail, and can be removed or deprecated in future Python releases. It's recommended to use public APIs from the *inspect* module for any introspection needs.

30.13.9 Interface de Linha de Comando

The *inspect* module also provides a basic introspection capability from the command line.

By default, accepts the name of a module and prints the source of that module. A class or function within the module can be printed instead by appended a colon and the qualified name of the target object.

`--details`
Print information about the specified object rather than the source code

30.14 `site` — Gancho de configuração específico do site

Código Fonte: Lib/site.py

Este módulo é importado automaticamente durante a inicialização. A importação automática pode ser suprimida usando a opção `-S` do interpretador.

A importação deste módulo anexará os caminhos específicos do site ao caminho de pesquisa do módulo e adicionará alguns recursos internos, a menos que `-S` tenha sido usada. Nesse caso, este módulo pode ser importado com segurança, sem modificações automáticas no caminho de pesquisa do módulo ou adições aos componentes internos. Para acionar explicitamente as adições habituais específicas do site, chame a função *site.main()*.

Alterado na versão 3.3: A importação do módulo usado para acionar a manipulação de caminhos, mesmo ao usar `-S`.

Começa construindo até quatro diretórios a partir de uma parte inicial e outra final. Para a parte inicial, ele usa `sys.prefix` and `sys.exec_prefix`; inícios vazios são pulados. Para a parte final, ele usa a string vazia e depois `lib/site-packages` (no Windows) ou `lib/python`*X.Y*`/site-packages` (no Unix e no Macintosh). Para cada uma das combinações distintas de parte final, ele vê se se refere a um diretório existente e, se for o caso, o adiciona ao `sys.path` e também inspeciona o novo caminho adicionado para os arquivos de configuração.

Alterado na versão 3.5: Suporte para o diretório "site-python" foi removido.

If a file named "pyvenv.cfg" exists one directory above sys.executable, sys.prefix and sys.exec_prefix are set to that directory and it is also checked for site-packages (sys.base_prefix and sys.base_exec_prefix will always be the "real" prefixes of the Python installation). If "pyvenv.cfg" (a bootstrap configuration file) contains the key "include-system-site-packages"

set to anything other than "false" (case-insensitive), the system-level prefixes will still also be searched for site-packages; otherwise they won't.

Um arquivo de configuração de caminho é aquele cujo nome tem o formato *name*`.pth` e que existe em um dos quatro diretórios mencionados acima; seu conteúdo são itens adicionais (um por linha) a serem adicionados ao `sys.path`. Itens inexistentes nunca são adicionados ao `sys.path` e não é verificado se o item se refere a um diretório, e não a um arquivo. Nenhum item é adicionado ao `sys.path` mais de uma vez. Linhas em branco e linhas iniciadas com `#` são ignoradas. Linhas iniciadas com `import` (seguidas de espaço ou tabulação) são executadas.

Por exemplo, suponha que `sys.prefix` e `sys.exec_prefix` sejam definidos com `/usr/local`. A biblioteca Python X.Y é instalado em `/usr/local/lib/python`*X.Y*. Suponha que isso tenha um subdiretório `/usr/local/lib/python`*X.Y*`/site-packages` com três subsubdiretórios, `foo`, `bar` e `spam`, e dois caminhos arquivos de configuração, `foo.pth` e `bar.pth`. Presuma que `foo.pth` contém o seguinte:

```
# foo package configuration

foo
bar
bletch
```

e que `bar.pth` contém:

```
# bar package configuration

bar
```

Em seguida, os seguintes diretórios específicos da versão são adicionados a `sys.path`, nesta ordem:

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

Observe que `bletch` é omitido porque não existe; o diretório `bar` precede o diretório `foo` porque `bar.pth` vem em ordem alfabética antes de `foo.pth`; e `spam` é omitido porque não é mencionado em nenhum dos arquivos de configuração de caminho.

After these path manipulations, an attempt is made to import a module named `sitecustomize`, which can perform arbitrary site-specific customizations. It is typically created by a system administrator in the site-packages directory. If this import fails with an *ImportError* or its subclass exception, and the exception's `name` attribute equals to `'sitecustomize'`, it is silently ignored. If Python is started without output streams available, as with `pythonw.exe` on Windows (which is used by default to start IDLE), attempted output from `sitecustomize` is ignored. Any other exception causes a silent and perhaps mysterious failure of the process.

After this, an attempt is made to import a module named `usercustomize`, which can perform arbitrary user-specific customizations, if *ENABLE_USER_SITE* is true. This file is intended to be created in the user site-packages directory (see below), which is part of `sys.path` unless disabled by `-s`. If this import fails with an *ImportError* or its subclass exception, and the exception's `name` attribute equals to `'usercustomize'`, it is silently ignored.

Note that for some non-Unix systems, `sys.prefix` and `sys.exec_prefix` are empty, and the path manipulations are skipped; however the import of `sitecustomize` and `usercustomize` is still attempted.

30.14.1 Configuração Readline

On systems that support *readline*, this module will also import and configure the *rlcompleter* module, if Python is started in interactive mode and without the `-S` option. The default behavior is enable tab-completion and to use `~/.python_history` as the history save file. To disable it, delete (or override) the *sys.__interactivehook__* attribute in your `sitecustomize` or `usercustomize` module or your `PYTHONSTARTUP` file.

Alterado na versão 3.4: Activation of rlcompleter and history was made automatic.

30.14.2 Conteúdo do módulo

`site.`**`PREFIXES`**
A list of prefixes for site-packages directories.

`site.`**`ENABLE_USER_SITE`**
Flag showing the status of the user site-packages directory. `True` means that it is enabled and was added to `sys.path`. `False` means that it was disabled by user request (with `-s` or `PYTHONNOUSERSITE`). `None` means it was disabled for security reasons (mismatch between user or group id and effective id) or by an administrator.

`site.`**`USER_SITE`**
Path to the user site-packages for the running Python. Can be `None` if *getusersitepackages()* hasn't been called yet. Default value is `~/.local/lib/python`*X.Y*`/site-packages` for UNIX and non-framework Mac OS X builds, `~/Library/Python/`*X.Y*`/lib/python/site-packages` for Mac framework builds, and *%APPDATA%*`\Python\Python`*XY*`\site-packages` on Windows. This directory is a site directory, which means that `.pth` files in it will be processed.

`site.`**`USER_BASE`**
Path to the base directory for the user site-packages. Can be `None` if *getuserbase()* hasn't been called yet. Default value is `~/.local` for UNIX and Mac OS X non-framework builds, `~/Library/Python/`*X.Y* for Mac framework builds, and *%APPDATA%*`\Python` for Windows. This value is used by Distutils to compute the installation directories for scripts, data files, Python modules, etc. for the user installation scheme. See also `PYTHONUSERBASE`.

`site.`**`main`**`()`
Adds all the standard site-specific directories to the module search path. This function is called automatically when this module is imported, unless the Python interpreter was started with the `-S` flag.

Alterado na versão 3.3: This function used to be called unconditionally.

`site.`**`addsitedir`**`(`*sitedir*, *known_paths=None*`)`
Add a directory to sys.path and process its `.pth` files. Typically used in `sitecustomize` or `usercustomize` (see above).

`site.`**`getsitepackages`**`()`
Return a list containing all global site-packages directories.

Novo na versão 3.2.

`site.`**`getuserbase`**`()`
Return the path of the user base directory, *USER_BASE*. If it is not initialized yet, this function will also set it, respecting `PYTHONUSERBASE`.

Novo na versão 3.2.

`site.`**`getusersitepackages`**`()`
Return the path of the user-specific site-packages directory, *USER_SITE*. If it is not initialized yet, this function will also set it, respecting `PYTHONNOUSERSITE` and *USER_BASE*.

Novo na versão 3.2.

30.14.3 Interface de Linha de Comando

The *site* module also provides a way to get the user directories from the command line:

```
$ python3 -m site --user-site
/home/user/.local/lib/python3.3/site-packages
```

If it is called without arguments, it will print the contents of *sys.path* on the standard output, followed by the value of *USER_BASE* and whether the directory exists, then the same thing for *USER_SITE*, and finally the value of *ENABLE_USER_SITE*.

--user-base
: Print the path to the user base directory.

--user-site
: Print the path to the user site-packages directory.

If both options are given, user base and user site will be printed (always in this order), separated by *os.pathsep*.

If any option is given, the script will exit with one of these values: `0` if the user site-packages directory is enabled, `1` if it was disabled by the user, `2` if it is disabled for security reasons or by an administrator, and a value greater than 2 if there is an error.

Ver também:

PEP 370 – Diretório site-packages por usuário.

CAPÍTULO 31

Interpretadores Python Personalizados

Os módulos descritos neste capítulo permitem a escrita de interfaces semelhantes ao intérprete interativo da Python. Se você quer um intérprete de Python que suporte algum recurso especial, além do idioma de Python, você deve olhar para o módulo: mod: *code*. (O módulo mod: *codeop* é de nível inferior, usado para suportar compilação de um pedaço possivelmente incompleto do código Python).

A lista completa de módulos descritos neste capítulo é:

31.1 code — Classes Bases do Intérprete

Código Fonte: Lib/code.py

O módulo `code` fornece facilidades para implementarmos loops de leitura-eval-escrita no código Python. São incluídas duas classes e funções de conveniência que podem ser usadas para criar aplicativos que fornecem um prompt de interpretação interativa.

class `code.`**InteractiveInterpreter**(*locals=None*)
Esta classe trata de analisar e interpretar o estado (espaço de nome do usuário); o mesmo não lida com buffer de entrada ou solicitação ou nomeação de arquivo de entrada (o nome do arquivo é sempre passado explicitamente). O argumento opcional *local* especifica o dicionário no qual o código será executado; ele é padrão para um dicionário recém-criado com a chave `'__name__'` set to `'__console__'` and key `'__doc__'` definido com `None`.

class `code.`**InteractiveConsole**(*locals=None*, *filename="<console>"*)
Closely emulate the behavior of the interactive Python interpreter. This class builds on *`InteractiveInterpreter`* and adds prompting using the familiar `sys.ps1` and `sys.ps2`, and input buffering.

`code.`**interact**(*banner=None*, *readfunc=None*, *local=None*, *exitmsg=None*)
Convenience function to run a read-eval-print loop. This creates a new instance of *`InteractiveConsole`* and sets *readfunc* to be used as the *`InteractiveConsole.raw_input()`* method, if provided. If *local* is provided, it is passed to the *`InteractiveConsole`* constructor for use as the default namespace for the

interpreter loop. The *interact()* method of the instance is then run with *banner* and *exitmsg* passed as the banner and exit message to use, if provided. The console object is discarded after use.

Alterado na versão 3.6: Parâmetro adicionado *exitmsg*.

code.**compile_command**(*source*, *filename="<input>"*, *symbol="single"*)

This function is useful for programs that want to emulate Python's interpreter main loop (a.k.a. the read-eval-print loop). The tricky part is to determine when the user has entered an incomplete command that can be completed by entering more text (as opposed to a complete command or a syntax error). This function *almost* always makes the same decision as the real interpreter main loop.

source is the source string; *filename* is the optional filename from which source was read, defaulting to `'<input>'`; and *symbol* is the optional grammar start symbol, which should be `'single'` (the default), `'eval'` or `'exec'`.

Returns a code object (the same as `compile(source, filename, symbol)`) if the command is complete and valid; `None` if the command is incomplete; raises *SyntaxError* if the command is complete and contains a syntax error, or raises *OverflowError* or *ValueError* if the command contains an invalid literal.

31.1.1 Objetos de Intérprete Interativo

InteractiveInterpreter.**runsource**(*source*, *filename="<input>"*, *symbol="single"*)

Compile and run some source in the interpreter. Arguments are the same as for *compile_command()*; the default for *filename* is `'<input>'`, and for *symbol* is `'single'`. One of several things can happen:

- The input is incorrect; *compile_command()* raised an exception (*SyntaxError* or *OverflowError*). A syntax traceback will be printed by calling the *showsyntaxerror()* method. *runsource()* returns `False`.
- The input is incomplete, and more input is required; *compile_command()* returned `None`. *runsource()* returns `True`.
- The input is complete; *compile_command()* returned a code object. The code is executed by calling the *runcode()* (which also handles run-time exceptions, except for *SystemExit*). *runsource()* returns `False`.

The return value can be used to decide whether to use `sys.ps1` or `sys.ps2` to prompt the next line.

InteractiveInterpreter.**runcode**(*code*)

Execute a code object. When an exception occurs, *showtraceback()* is called to display a traceback. All exceptions are caught except *SystemExit*, which is allowed to propagate.

A note about *KeyboardInterrupt*: this exception may occur elsewhere in this code, and may not always be caught. The caller should be prepared to deal with it.

InteractiveInterpreter.**showsyntaxerror**(*filename=None*)

Display the syntax error that just occurred. This does not display a stack trace because there isn't one for syntax errors. If *filename* is given, it is stuffed into the exception instead of the default filename provided by Python's parser, because it always uses `'<string>'` when reading from a string. The output is written by the *write()* method.

InteractiveInterpreter.**showtraceback**()

Display the exception that just occurred. We remove the first stack item because it is within the interpreter object implementation. The output is written by the *write()* method.

Alterado na versão 3.5: The full chained traceback is displayed instead of just the primary traceback.

InteractiveInterpreter.**write**(*data*)

Write a string to the standard error stream (`sys.stderr`). Derived classes should override this to provide the appropriate output handling as needed.

31.1.2 Objetos de Console Interativos

The *InteractiveConsole* class is a subclass of *InteractiveInterpreter*, and so offers all the methods of the interpreter objects as well as the following additions.

`InteractiveConsole.`**`interact`**(*banner=None*, *exitmsg=None*)
Closely emulate the interactive Python console. The optional *banner* argument specify the banner to print before the first interaction; by default it prints a banner similar to the one printed by the standard Python interpreter, followed by the class name of the console object in parentheses (so as not to confuse this with the real interpreter – since it's so close!).

The optional *exitmsg* argument specifies an exit message printed when exiting. Pass the empty string to suppress the exit message. If *exitmsg* is not given or `None`, a default message is printed.

Alterado na versão 3.4: To suppress printing any banner, pass an empty string.

Alterado na versão 3.6: Imprima uma mensagem de saída ao sair.

`InteractiveConsole.`**`push`**(*line*)
Push a line of source text to the interpreter. The line should not have a trailing newline; it may have internal newlines. The line is appended to a buffer and the interpreter's `runsource()` method is called with the concatenated contents of the buffer as source. If this indicates that the command was executed or invalid, the buffer is reset; otherwise, the command is incomplete, and the buffer is left as it was after the line was appended. The return value is `True` if more input is required, `False` if the line was dealt with in some way (this is the same as `runsource()`).

`InteractiveConsole.`**`resetbuffer`**()
Remove any unhandled source text from the input buffer.

`InteractiveConsole.`**`raw_input`**(*prompt=""*)
Write a prompt and read a line. The returned line does not include the trailing newline. When the user enters the EOF key sequence, *EOFError* is raised. The base implementation reads from `sys.stdin`; a subclass may replace this with a different implementation.

31.2 codeop — Compilar código Python

Código Fonte: Lib/codeop.py

O módulo *codeop* fornece utilitários sobre os quais o loop de leitura-execução-exibição do Python pode ser emulado, como é feito no módulo *code*. Como resultado, você provavelmente não deseja usar o módulo diretamente; se você deseja incluir tal loop em seu programa, você provavelmente deseja usar o módulo *code*.

Há duas partes para esta tarefa:

1. Ser capaz de dizer se uma linha de entrada completa uma instrução Python: em suma, dizer se deve exibir '>>>' ou '...' em seguida.
2. Lembrar quais instruções futuras o usuário inseriu, para que as entradas subsequentes possam ser compiladas com essas declarações em vigor.

O módulo *codeop* fornece uma maneira de fazer cada uma dessas coisas e uma maneira de fazer as duas coisas.

Para fazer apenas a primeira:

`codeop.`**`compile_command`**(*source*, *filename="<input>"*, *symbol="single"*)
Tenta compilar *source*, que deve ser uma string de código Python e retornar um objeto código se *source* for um código Python válido. Nesse caso, o atributo de nome de arquivo do objeto código será *filename*, cujo padrão é `'<input>'`. Retorna `None` se *source não* é um código Python válido, mas é um prefixo de código Python válido.

Se houver um problema com *source*, uma exceção será levantada. *SyntaxError* é levantada se houver sintaxe Python inválida, e *OverflowError* ou *ValueError* se houver um literal inválido.

The *symbol* argument determines whether *source* is compiled as a statement (`'single'`, the default), as a sequence of statements (`'exec'`) or as an *expression* (`'eval'`). Any other value will cause *ValueError* to be raised.

Nota: É possível (mas não provável) que o analisador sintático pare de analisar com um resultado bem-sucedido antes de chegar ao final da fonte; neste caso, os símbolos finais podem ser ignorados em vez de causar um erro. Por exemplo, uma barra invertida seguida por duas novas linhas pode ser seguida por lixo arbitrário. Isso será corrigido quando a API para o analisador for melhor.

---

class codeop.**Compile**

Instâncias desta classe têm métodos `__call__()` idênticos em assinatura à função embutida *compile()*, mas com a diferença de que se a instância compilar o texto do programa contendo uma instrução *__future__*, a instância se "lembra" e compila todos os textos de programa subsequentes com a instrução em vigor.

class codeop.**CommandCompiler**

Instâncias desta classe têm métodos `__call__()` idênticos em assinatura a *compile_command()*; a diferença é que se a instância compila o texto do programa contendo uma instrução `__future__`, a instância se "lembra" e compila todos os textos do programa subsequentes com a instrução em vigor.

CAPÍTULO 32

Importando Módulos

Os módulos descritos neste capítulo fornecem novas maneiras de importar outros módulos Python e hooks para personalizar o processo de importação.

A lista completa de módulos descritos neste capítulo é:

32.1 `zipimport` — Import modules from Zip archives

This module adds the ability to import Python modules (`*.py`, `*.pyc`) and packages from ZIP-format archives. It is usually not needed to use the *`zipimport`* module explicitly; it is automatically used by the built-in `import` mechanism for *`sys.path`* items that are paths to ZIP archives.

Typically, *`sys.path`* is a list of directory names as strings. This module also allows an item of *`sys.path`* to be a string naming a ZIP file archive. The ZIP archive can contain a subdirectory structure to support package imports, and a path within the archive can be specified to only import from a subdirectory. For example, the path `example.zip/lib/` would only import from the `lib/` subdirectory within the archive.

Any files may be present in the ZIP archive, but only files `.py` and `.pyc` are available for import. ZIP import of dynamic modules (`.pyd`, `.so`) is disallowed. Note that if an archive only contains `.py` files, Python will not attempt to modify the archive by adding the corresponding `.pyc` file, meaning that if a ZIP archive doesn't contain `.pyc` files, importing may be rather slow.

ZIP archives with an archive comment are currently not supported.

Ver também:

PKZIP Application Note Documentação do formato de arquivo ZIP feita por Phil Katz, criador do formato e dos algoritmos usados.

PEP 273 - Importar módulos de arquivos Zip Written by James C. Ahlstrom, who also provided an implementation. Python 2.3 follows the specification in PEP 273, but uses an implementation written by Just van Rossum that uses the import hooks described in PEP 302.

PEP 302 - New Import Hooks The PEP to add the import hooks that help this module work.

This module defines an exception:

exception zipimport.**ZipImportError**
Exception raised by zipimporter objects. It's a subclass of *ImportError*, so it can be caught as *ImportError*, too.

32.1.1 zipimporter Objects

zipimporter is the class for importing ZIP files.

class zipimport.**zipimporter**(*archivepath*)
Create a new zipimporter instance. *archivepath* must be a path to a ZIP file, or to a specific path within a ZIP file. For example, an *archivepath* of `foo/bar.zip/lib` will look for modules in the `lib` directory inside the ZIP file `foo/bar.zip` (provided that it exists).

ZipImportError is raised if *archivepath* doesn't point to a valid ZIP archive.

find_module(*fullname*[, *path*])
Search for a module specified by *fullname*. *fullname* must be the fully qualified (dotted) module name. It returns the zipimporter instance itself if the module was found, or *None* if it wasn't. The optional *path* argument is ignored—it's there for compatibility with the importer protocol.

get_code(*fullname*)
Return the code object for the specified module. Raise *ZipImportError* if the module couldn't be found.

get_data(*pathname*)
Return the data associated with *pathname*. Raise *OSError* if the file wasn't found.

Alterado na versão 3.3: *IOError* costumava ser levantado em vez do *OSError*.

get_filename(*fullname*)
Return the value `__file__` would be set to if the specified module was imported. Raise *ZipImportError* if the module couldn't be found.

Novo na versão 3.1.

get_source(*fullname*)
Return the source code for the specified module. Raise *ZipImportError* if the module couldn't be found, return *None* if the archive does contain the module, but has no source for it.

is_package(*fullname*)
Devolve `True` se o módulo especificado por *fullname* é um pacote. Levanta *ZipImportError* se o módulo não pode ser localizado.

load_module(*fullname*)
Load the module specified by *fullname*. *fullname* must be the fully qualified (dotted) module name. It returns the imported module, or raises *ZipImportError* if it wasn't found.

archive
The file name of the importer's associated ZIP file, without a possible subpath.

prefix
The subpath within the ZIP file where modules are searched. This is the empty string for zipimporter objects which point to the root of the ZIP file.

The *archive* and *prefix* attributes, when combined with a slash, equal the original *archivepath* argument given to the *zipimporter* constructor.

32.1.2 Exemplos

Here is an example that imports a module from a ZIP archive - note that the *zipimport* module is not explicitly used.

```
$ unzip -l example.zip
Archive:  example.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
     8467  11-26-02 22:30   jwzthreading.py
 --------                   -------
     8467                   1 file
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, 'example.zip')  # Add .zip file to front of path
>>> import jwzthreading
>>> jwzthreading.__file__
'example.zip/jwzthreading.py'
```

32.2 `pkgutil` — Utilitário de extensão de pacote

Código Fonte: Lib/pkgutil.py

Este módulo fornece utilitários para o sistema de importação, em particular suporte a pacotes.

class pkgutil.**ModuleInfo**(*module_finder*, *name*, *ispkg*)
Um namedtuple que contém um breve resumo das informações de um módulo.

Novo na versão 3.6.

pkgutil.**extend_path**(*path*, *name*)
Estende o caminho de pesquisa para os módulos que compõem um pacote. O uso pretendido é colocar o seguinte código no `__init__.py` de um pacote:

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

This will add to the package's `__path__` all subdirectories of directories on `sys.path` named after the package. This is useful if one wants to distribute different parts of a single logical package as multiple directories.

It also looks for `*.pkg` files beginning where `*` matches the *name* argument. This feature is similar to `*.pth` files (see the *site* module for more information), except that it doesn't special-case lines starting with `import`. A `*.pkg` file is trusted at face value: apart from checking for duplicates, all entries found in a `*.pkg` file are added to the path, regardless of whether they exist on the filesystem. (This is a feature.)

If the input path is not a list (as is the case for frozen packages) it is returned unchanged. The input path is not modified; an extended copy is returned. Items are only appended to the copy at the end.

It is assumed that *sys.path* is a sequence. Items of *sys.path* that are not strings referring to existing directories are ignored. Unicode items on *sys.path* that cause errors when used as filenames may cause this function to raise an exception (in line with *os.path.isdir()* behavior).

class pkgutil.**ImpImporter**(*dirname=None*)
PEP 302 Finder that wraps Python's "classic" import algorithm.

If *dirname* is a string, a **PEP 302** finder is created that searches that directory. If *dirname* is `None`, a **PEP 302** finder is created that searches the current *sys.path*, plus any modules that are frozen or built-in.

Note that *ImpImporter* does not currently support being used by placement on *sys.meta_path*.

Obsoleto desde a versão 3.3: This emulation is no longer needed, as the standard import mechanism is now fully PEP 302 compliant and available in *importlib*.

class pkgutil.**ImpLoader**(*fullname*, *file*, *filename*, *etc*)
Loader that wraps Python's "classic" import algorithm.

Obsoleto desde a versão 3.3: This emulation is no longer needed, as the standard import mechanism is now fully PEP 302 compliant and available in *importlib*.

pkgutil.**find_loader**(*fullname*)
Retrieve a module *loader* for the given *fullname*.

This is a backwards compatibility wrapper around *importlib.util.find_spec()* that converts most failures to *ImportError* and only returns the loader rather than the full `ModuleSpec`.

Alterado na versão 3.3: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

Alterado na versão 3.4: Updated to be based on **PEP 451**

pkgutil.**get_importer**(*path_item*)
Retrieve a *finder* for the given *path_item*.

The returned finder is cached in *sys.path_importer_cache* if it was newly created by a path hook.

The cache (or part of it) can be cleared manually if a rescan of *sys.path_hooks* is necessary.

Alterado na versão 3.3: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

pkgutil.**get_loader**(*module_or_name*)
Get a *loader* object for *module_or_name*.

If the module or package is accessible via the normal import mechanism, a wrapper around the relevant part of that machinery is returned. Returns `None` if the module cannot be found or imported. If the named module is not already imported, its containing package (if any) is imported, in order to establish the package `__path__`.

Alterado na versão 3.3: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

Alterado na versão 3.4: Updated to be based on **PEP 451**

pkgutil.**iter_importers**(*fullname=''*)
Yield *finder* objects for the given module name.

If fullname contains a '.', the finders will be for the package containing fullname, otherwise they will be all registered top level finders (i.e. those on both sys.meta_path and sys.path_hooks).

If the named module is in a package, that package is imported as a side effect of invoking this function.

If no module name is specified, all top level finders are produced.

Alterado na versão 3.3: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

pkgutil.**iter_modules**(*path=None*, *prefix=''*)
Yields *ModuleInfo* for all submodules on *path*, or, if *path* is `None`, all top-level modules on `sys.path`.

path should be either `None` or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

Nota: Only works for a *finder* which defines an `iter_modules()` method. This interface is non-standard, so the module also provides implementations for *importlib.machinery.FileFinder* and *zipimport.zipimporter*.

Alterado na versão 3.3: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

`pkgutil.`**`walk_packages`**(*path=None*, *prefix=''*, *onerror=None*)

Yields *ModuleInfo* for all modules recursively on *path*, or, if *path* is `None`, all accessible modules.

path should be either `None` or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

Note that this function must import all *packages* (*not* all modules!) on the given *path*, in order to access the `__path__` attribute to find submodules.

onerror is a function which gets called with one argument (the name of the package which was being imported) if any exception occurs while trying to import a package. If no *onerror* function is supplied, *ImportError*s are caught and ignored, while all other exceptions are propagated, terminating the search.

Exemplos:

```
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ + '.')
```

Nota: Only works for a *finder* which defines an `iter_modules()` method. This interface is non-standard, so the module also provides implementations for *importlib.machinery.FileFinder* and *zipimport.zipimporter*.

Alterado na versão 3.3: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

`pkgutil.`**`get_data`**(*package*, *resource*)

Get a resource from a package.

This is a wrapper for the *loader* *get_data* API. The *package* argument should be the name of a package, in standard module format (`foo.bar`). The *resource* argument should be in the form of a relative filename, using `/` as the path separator. The parent directory name `..` is not allowed, and nor is a rooted name (starting with a `/`).

The function returns a binary string that is the contents of the specified resource.

For packages located in the filesystem, which have already been imported, this is the rough equivalent of:

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()
```

If the package cannot be located or loaded, or it uses a *loader* which does not support *get_data*, then `None` is returned. In particular, the *loader* for *namespace packages* does not support *get_data*.

32.3 `modulefinder` — Procurar módulos usados por um script

Código Fonte: Lib/modulefinder.py

Este módulo fornece uma classe: class: *ModuleFinder* que pode ser usada para determinar o conjunto de módulos importados por um script. O " modulefinder.py" também pode ser executado como um script, fornecendo o nome do arquivo de um script Python como seu argumento, após o qual um relatório dos módulos importados será impresso.

`modulefinder.`**`AddPackagePath`**(*pkg_name*, *path*)
Registre que o pacote chamado * pkg_name * pode ser encontrado no caminho * especificado *

`modulefinder.`**`ReplacePackage`**(*oldname*, *newname*)
Permite especificar que o módulo chamado * oldname * é de fato o pacote chamado * newname *.

`class` `modulefinder.`**`ModuleFinder`**(*path=None*, *debug=0*, *excludes=[]*, *replace_paths=[]*)
Esta classe fornece os métodos: meth: *run_script* e: meth:' report' para determinar o conjunto de módulos importados por um script. * path * pode ser uma lista de diretórios para procurar por módulos; se não especificado, " sys.path" é usado. * debug * define o nível de depuração; valores mais altos fazem a classe imprimir mensagens de depuração sobre o que está fazendo. * excludes * é uma lista de nomes de módulos a serem excluídos da análise. * replace_paths * é uma lista de tuplas " (oldpath, newpath) " que serão substituídas nos caminhos dos módulos.

`report`()
Imprima um relatório na saída padrão que lista os módulos importados pelo script e seus caminhos, bem como os módulos que estão faltando ou parecem estar ausentes.

`run_script`(*pathname*)
Analise o conteúdo do arquivo * pathname *, que deve conter o código Python.

`modules`
Um nome de módulo de mapeamento de dicionário para módulos. Veja: ref: *modulefinder-example*.

32.3.1 Exemplo de uso de: class: *ModuleFinder*

O script que será analisado posteriormente (bacon.py)

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

O script que irá gerar o relatório de bacon.py

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print('Loaded modules:')
```

(continua na próxima página)

(continuação da página anterior)

```
for name, mod in finder.modules.items():
    print('%s: ' % name, end='')
    print(','.join(list(mod.globalnames.keys())[:3]))

print('-'*50)
print('Modules not imported:')
print('\n'.join(finder.badmodules.keys()))
```

Saída de amostra (pode variar dependendo da arquitetura)

```
Loaded modules:
_types:
copyreg:  _inverted_registry,_slotnames,__all__
sre_compile:  isstring,_sre,_optimize_unicode
_sre:
sre_constants:  REPEAT_ONE,makedict,AT_END_LINE
sys:
re:  __module__,finditer,_expand
itertools:
__main__:  re,itertools,baconhameggs
sre_parse:  _PATTERNENDERS,SRE_FLAG_UNICODE
array:
types:  __module__,IntType,TypeType
-----------------------------------------------------
Modules not imported:
guido.python.ham
baconhameggs
```

32.4 `runpy` — Localizando e executando módulos Python

Código Fonte: Lib/runpy.py

O módulo *runpy* é usado para localizar e executar módulos Python sem importá-los primeiro. Seu principal uso é implementar a opção de linha de comando `-m` que permite que os scripts sejam localizados usando o espaço de nomes do módulo Python em vez do sistema de arquivos.

Observe que este *não* é um módulo sandbox - todo o código é executado no processo atual, e quaisquer efeitos colaterais (como importações em cache de outros módulos) irão permanecer em vigor após o retorno da função.

Além disso, quaisquer funções e classes definidas pelo código executado não têm garantia de funcionar corretamente após o retorno de uma função *runpy*. Se essa limitação não for aceitável para um determinado caso de uso, *importlib* provavelmente será uma escolha mais adequada do que este módulo.

O módulo *runpy* fornece duas funções:

runpy.**run_module**(*mod_name*, *init_globals=None*, *run_name=None*, *alter_sys=False*)

Execute o código do módulo especificado e retorne o dicionário global do módulo resultante. O código do módulo é localizado primeiro usando o mecanismo de importação padrão (consulte **PEP 302** para detalhes) e então executado em um novo espaço de nomes de módulo.

O argumento *mod_name* deve ser um nome de módulo absoluto. Se o nome do módulo se referir a um pacote ao invés de um módulo normal, então esse pacote é importado e o submódulo `__main__` dentro desse pacote é então executado e o dicionário global do módulo resultante retornado.

O argumento opcional de dicionário *init_globals* pode ser usado para preencher previamente o dicionário global do módulo antes do código ser executado. O dicionário fornecido não será alterado. Se qualquer uma das variáveis globais especiais abaixo for definida no dicionário fornecido, estas definições serão substituídas por *run_module()*.

As variáveis globais especiais __name__, __spec__, __file__, __cached__, __loader__ e __package__ são definidas no dicionário global antes que o código do módulo seja executado (Observe que este é um conjunto mínimo de variáveis - outras variáveis podem ser definidas implicitamente como um detalhe de implementação do interpretador).

__name__ é definido como *run_name* se este argumento opcional não for *None*, para mod_name + '.__main__' se o módulo nomeado for um pacote e para o argumento *mod_name* caso contrário .

__spec__ será definido adequadamente para o modo *efetivamente* importado (isto é, __spec__.name vai sempre ser *mod_name* ou mod_name + '.__main__, nunca *run_name*).

__file__, __cached__, __loader__ e __package__ são definidos como normal com base na especificação do módulo.

Se o argumento *alter_sys* for fornecido e for avaliado como *True*, então sys.argv[0] será atualizado com o valor de __file__ e sys.modules[__name__] é atualizado com um objeto de módulo temporário para o módulo que está sendo executado. Ambos sys.argv[0] e sys.modules[__name__] são restaurados para seus valores originais antes que a função retorne.

Note que esta manipulação de *sys* não é segura para thread. Outras threads podem ver o módulo parcialmente inicializado, bem como a lista alterada de argumentos. É recomendado que o módulo *sys* seja deixado sozinho ao invocar esta função a partir do código encadeado.

Ver também:

A opção -m oferece funcionalidade equivalente na linha de comando.

Alterado na versão 3.1: Adicionada capacidade de executar pacotes procurando por um sub-módulo __main__.

Alterado na versão 3.2: Adicionada a variável global __cached__ (veja **PEP 3147**).

Alterado na versão 3.4: Atualizado para aproveitar o recurso de especificação do módulo adicionado por **PEP 451**. Isso permite que __cached__ seja configurado corretamente para módulos executados desta forma, assim como garante que o nome real do módulo esteja sempre acessível como __spec__.name.

runpy.**run_path**(*file_path*, *init_globals=None*, *run_name=None*)

Executa o código no local do sistema de arquivos nomeado e retorna o dicionário global do módulo resultante. Assim como um nome de script fornecido à linha de comando CPython, o caminho fornecido pode se referir a um arquivo de origem Python, um arquivo de bytecode compilado ou uma entrada sys.path válida contendo um módulo __main__ (por exemplo, um arquivo zip contendo um arquivo de topo de nível __main__.py).

Para um script simples, o código especificado é simplesmente executado em um novo espaço de nomes de módulo. Para uma entrada sys.path válida (normalmente um arquivo zip ou diretório), a entrada é primeiro adicionada ao início de sys.path. A função então procura e executa um módulo *__main__* usando o caminho atualizado. Observe que não há proteção especial contra invocar uma entrada *__main__* existente localizada em outro lugar em sys.path se não houver tal módulo no local especificado.

O argumento opcional de dicionário *init_globals* pode ser usado para preencher previamente o dicionário global do módulo antes do código ser executado. O dicionário fornecido não será alterado. Se qualquer uma das variáveis globais especiais abaixo for definida no dicionário fornecido, estas definições serão substituídas por *run_path()*.

As variáveis globais especiais __name__, __spec__, __file__, __cached__, __loader__ e __package__ são definidas no dicionário global antes que o código do módulo seja executado (Observe que este é um conjunto mínimo de variáveis - outras variáveis podem ser definidas implicitamente como um detalhe de implementação do interpretador).

__name__ é definido como *run_name* se este argumento opcional não for *None* e como '<run_path>' caso contrário.

Se o caminho fornecido referenciar diretamente um arquivo de script (seja como fonte ou como bytecode pré-compilado), então __file__ será definido para o caminho fornecido e __spec__, __cached__, __loader__ `` e ``__package__ serão todos definidos como *None*.

Se o caminho fornecido for uma referência a uma entrada sys.path válida, então __spec__ será definido apropriadamente para o módulo __main__ importado (ou seja, __spec__.name sempre será " __principal__"). __file__, __cached__, __loader__ e __package__ serão definidos como normal com base na especificação do módulo.

A number of alterations are also made to the *sys* module. Firstly, sys.path may be altered as described above. sys.argv[0] is updated with the value of file_path and sys.modules[__name__] is updated with a temporary module object for the module being executed. All modifications to items in *sys* are reverted before the function returns.

Note que, diferentemente de *run_module()*, as alterações feitas em *sys* não são opcionais nesta função, pois esses ajustes são essenciais para permitir a execução das entradas do sys.path. Como as limitações de segurança de encadeamento ainda se aplicam, o uso dessa função no código encadeado deve ser serializado com o bloqueio de importação ou delegado a um processo separado.

Ver também:

using-on-interface-options para funcionalidade equivalente na linha de comando (python path/to/script).

Novo na versão 3.2.

Alterado na versão 3.4: Atualizado para aproveitar o recurso de especificação do módulo adicionado por **PEP 451**. Isso permite que __cached__ seja definido corretamente no caso em que __main__ é importado de uma entrada sys.path válida em vez de ser executado diretamente.

Ver também:

PEP 338 – Executando módulos como scripts PEP escrita e implementada por Nick Coghlan.

PEP 366 – Importações relativas explícitas do módulo principal PEP escrita e implementada por Nick Coghlan.

PEP 451 – Um tipo ModuleSpec para o sistema de importação PEP escrita e implementada por Eric Snow

using-on-general - Detalhes da linha de comando do CPython

A função *importlib.import_module()*

32.5 importlib — The implementation of import

Novo na versão 3.1.

Código Fonte: Lib/importlib/__init__.py

32.5.1 Introdução

The purpose of the *importlib* package is two-fold. One is to provide the implementation of the import statement (and thus, by extension, the *__import__()* function) in Python source code. This provides an implementation of import which is portable to any Python interpreter. This also provides an implementation which is easier to comprehend than one implemented in a programming language other than Python.

Two, the components to implement import are exposed in this package, making it easier for users to create their own custom objects (known generically as an *importer*) to participate in the import process.

Ver também:

import A referência da linguagem para a instrução `import`

Especificação dos pacotes Original specification of packages. Some semantics have changed since the writing of this document (e.g. redirecting based on `None` in *`sys.modules`*).

A função *`__import__()`* The `import` statement is syntactic sugar for this function.

PEP 235 Import on Case-Insensitive Platforms

PEP 263 Defining Python Source Code Encodings

PEP 302 New Import Hooks

PEP 328 Importa: Multi-Linha e Absoluto/Relativo

PEP 366 Main module explicit relative imports

PEP 420 Implicit namespace packages

PEP 451 A ModuleSpec Type for the Import System

PEP 488 Eliminação de arquivos PYO

PEP 489 inicialização de módulo extensão Multi-fase

PEP 552 Deterministic pycs

PEP 3120 Usando UFT-8 como fonte padrão de codificação

PEP 3147 PYC Repository Directories

32.5.2 Funções

`importlib.`**`__import__`**(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)
An implementation of the built-in *`__import__()`* function.

> **Nota:** Programmatic importing of modules should use *`import_module()`* instead of this function.

`importlib.`**`import_module`**(*name*, *package=None*)
Import a module. The *name* argument specifies what module to import in absolute or relative terms (e.g. either `pkg.mod` or `..mod`). If the name is specified in relative terms, then the *package* argument must be set to the name of the package which is to act as the anchor for resolving the package name (e.g. `import_module('..mod', 'pkg.subpkg')` will import `pkg.mod`).

The *`import_module()`* function acts as a simplifying wrapper around *`importlib.__import__()`*. This means all semantics of the function are derived from *`importlib.__import__()`*. The most important difference between these two functions is that *`import_module()`* returns the specified package or module (e.g. `pkg.mod`), while *`__import__()`* returns the top-level package or module (e.g. `pkg`).

If you are dynamically importing a module that was created since the interpreter began execution (e.g., created a Python source file), you may need to call *`invalidate_caches()`* in order for the new module to be noticed by the import system.

Alterado na versão 3.3: Parent packages are automatically imported.

`importlib.`**`find_loader`**(*name*, *path=None*)
Find the loader for a module, optionally within the specified *path*. If the module is in *`sys.modules`*, then `sys.modules[name].__loader__` is returned (unless the loader would be `None` or is not set, in which case *`ValueError`* is raised). Otherwise a search using *`sys.meta_path`* is done. `None` is returned if no loader is found.

A dotted name does not have its parents implicitly imported as that requires loading them and that may not be desired. To properly import a submodule you will need to import all parent packages of the submodule and use the correct argument to *path*.

Novo na versão 3.3.

Alterado na versão 3.4: If `__loader__` is not set, raise *ValueError*, just like when the attribute is set to `None`.

Obsoleto desde a versão 3.4: Use *importlib.util.find_spec()* instead.

`importlib.`**`invalidate_caches`**`()`

Invalidate the internal caches of finders stored at *sys.meta_path*. If a finder implements `invalidate_caches()` then it will be called to perform the invalidation. This function should be called if any modules are created/installed while your program is running to guarantee all finders will notice the new module's existence.

Novo na versão 3.3.

`importlib.`**`reload`**`(`*module*`)`

Reload a previously imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (which can be different if re-importing causes a different object to be placed in *sys.modules*).

When *reload()* is executed:

- Python module's code is recompiled and the module-level code re-executed, defining a new set of objects which are bound to names in the module's dictionary by reusing the *loader* which originally loaded the module. The `init` function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects — with a `try` statement it can test for the table's presence and skip its initialization if desired:

```
try:
    cache
except NameError:
    cache = {}
```

It is generally not very useful to reload built-in or dynamically loaded modules. Reloading *sys*, *__main__*, *builtins* and other key modules is not recommended. In many cases extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from` … `import` …, calling *reload()* for the other module does not redefine the objects imported from it — one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (*module.name*) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

Novo na versão 3.4.

Alterado na versão 3.7: *ModuleNotFoundError* is raised when the module being reloaded lacks a ModuleSpec.

32.5.3 importlib.abc – Abstract base classes related to import

Codigo fonte: Lib/importlib/abc.py

The *importlib.abc* module contains all of the core abstract base classes used by import. Some subclasses of the core abstract base classes are also provided to help in implementing the core ABCs.

Hierarquia ABC:

```
object
 +-- Finder (deprecated)
 |    +-- MetaPathFinder
 |    +-- PathEntryFinder
 +-- Loader
      +-- ResourceLoader --------+
      +-- InspectLoader          |
           +-- ExecutionLoader --+
                                 +-- FileLoader
                                 +-- SourceLoader
```

class importlib.abc.**Finder**

An abstract base class representing a *finder*.

Obsoleto desde a versão 3.3: Use *MetaPathFinder* or *PathEntryFinder* instead.

abstractmethod find_module(*fullname*, *path=None*)

An abstract method for finding a *loader* for the specified module. Originally specified in **PEP 302**, this method was meant for use in *sys.meta_path* and in the path-based import subsystem.

Alterado na versão 3.4: Returns None when called instead of raising *NotImplementedError*.

class importlib.abc.**MetaPathFinder**

An abstract base class representing a *meta path finder*. For compatibility, this is a subclass of *Finder*.

Novo na versão 3.3.

find_spec(*fullname*, *path*, *target=None*)

An abstract method for finding a *spec* for the specified module. If this is a top-level import, *path* will be None. Otherwise, this is a search for a subpackage or module and *path* will be the value of __path__ from the parent package. If a spec cannot be found, None is returned. When passed in, target is a module object that the finder may use to make a more educated guess about what spec to return. *importlib.util.spec_from_loader()* may be useful for implementing concrete MetaPathFinders.

Novo na versão 3.4.

find_module(*fullname*, *path*)

A legacy method for finding a *loader* for the specified module. If this is a top-level import, *path* will be None. Otherwise, this is a search for a subpackage or module and *path* will be the value of __path__ from the parent package. If a loader cannot be found, None is returned.

If *find_spec()* is defined, backwards-compatible functionality is provided.

Alterado na versão 3.4: Returns None when called instead of raising *NotImplementedError*. Can use *find_spec()* to provide functionality.

Obsoleto desde a versão 3.4: Use *find_spec()* instead.

invalidate_caches()
An optional method which, when called, should invalidate any internal cache used by the finder. Used by *importlib.invalidate_caches()* when invalidating the caches of all finders on *sys.meta_path*.

Alterado na versão 3.4: Returns `None` when called instead of `NotImplemented`.

class importlib.abc.**PathEntryFinder**
An abstract base class representing a *path entry finder*. Though it bears some similarities to *MetaPathFinder*, `PathEntryFinder` is meant for use only within the path-based import subsystem provided by `PathFinder`. This ABC is a subclass of *Finder* for compatibility reasons only.

Novo na versão 3.3.

find_spec(*fullname*, *target=None*)
An abstract method for finding a *spec* for the specified module. The finder will search for the module only within the *path entry* to which it is assigned. If a spec cannot be found, `None` is returned. When passed in, `target` is a module object that the finder may use to make a more educated guess about what spec to return. *importlib.util.spec_from_loader()* may be useful for implementing concrete `PathEntryFinders`.

Novo na versão 3.4.

find_loader(*fullname*)
A legacy method for finding a *loader* for the specified module. Returns a 2-tuple of `(loader, portion)` where `portion` is a sequence of file system locations contributing to part of a namespace package. The loader may be `None` while specifying `portion` to signify the contribution of the file system locations to a namespace package. An empty list can be used for `portion` to signify the loader is not part of a namespace package. If `loader` is `None` and `portion` is the empty list then no loader or location for a namespace package were found (i.e. failure to find anything for the module).

If *find_spec()* is defined then backwards-compatible functionality is provided.

Alterado na versão 3.4: Returns `(None, [])` instead of raising *NotImplementedError*. Uses *find_spec()* when available to provide functionality.

Obsoleto desde a versão 3.4: Use *find_spec()* instead.

find_module(*fullname*)
A concrete implementation of *Finder.find_module()* which is equivalent to `self.find_loader(fullname)[0]`.

Obsoleto desde a versão 3.4: Use *find_spec()* instead.

invalidate_caches()
An optional method which, when called, should invalidate any internal cache used by the finder. Used by `PathFinder.invalidate_caches()` when invalidating the caches of all cached finders.

class importlib.abc.**Loader**
An abstract base class for a *loader*. See **PEP 302** for the exact definition for a loader.

Loaders that wish to support resource reading should implement a `get_resource_reader(fullname)` method as specified by *importlib.abc.ResourceReader*.

Alterado na versão 3.7: Introduced the optional `get_resource_reader()` method.

create_module(*spec*)
A method that returns the module object to use when importing a module. This method may return `None`, indicating that default module creation semantics should take place.

Novo na versão 3.4.

Alterado na versão 3.5: Starting in Python 3.6, this method will not be optional when *exec_module()* is defined.

exec_module(*module*)
An abstract method that executes the module in its own namespace when a module is imported or reloaded. The module should already be initialized when exec_module() is called. When this method exists, *create_module()* must be defined.

Novo na versão 3.4.

Alterado na versão 3.6: *create_module()* precisa ser definida.

load_module(*fullname*)
A legacy method for loading a module. If the module cannot be loaded, *ImportError* is raised, otherwise the loaded module is returned.

If the requested module already exists in *sys.modules*, that module should be used and reloaded. Otherwise the loader should create a new module and insert it into *sys.modules* before any loading begins, to prevent recursion from the import. If the loader inserted a module and the load fails, it must be removed by the loader from *sys.modules*; modules already in *sys.modules* before the loader began execution should be left alone (see *importlib.util.module_for_loader()*).

The loader should set several attributes on the module. (Note that some of these attributes can change when a module is reloaded):

- **__name__** O nome do módulo.
- **__file__** The path to where the module data is stored (not set for built-in modules).
- **__cached__** The path to where a compiled version of the module is/should be stored (not set when the attribute would be inappropriate).
- **__path__** A list of strings specifying the search path within a package. This attribute is not set on modules.
- **__package__** The parent package for the module/package. If the module is top-level then it has a value of the empty string. The *importlib.util.module_for_loader()* decorator can handle the details for __package__.
- **__loader__** The loader used to load the module. The *importlib.util.module_for_loader()* decorator can handle the details for __package__.

When *exec_module()* is available then backwards-compatible functionality is provided.

Alterado na versão 3.4: Raise *ImportError* when called instead of *NotImplementedError*. Functionality provided when *exec_module()* is available.

Obsoleto desde a versão 3.4: The recommended API for loading a module is *exec_module()* (and *create_module()*). Loaders should implement it instead of load_module(). The import machinery takes care of all the other responsibilities of load_module() when exec_module() is implemented.

module_repr(*module*)
A legacy method which when implemented calculates and returns the given module's repr, as a string. The module type's default repr() will use the result of this method as appropriate.

Novo na versão 3.3.

Alterado na versão 3.4: Made optional instead of an abstractmethod.

Obsoleto desde a versão 3.4: The import machinery now takes care of this automatically.

class importlib.abc.**ResourceReader**
Uma *classe base abstrata* para fornecer a capacidade de ler *recursos*.

Da perspectiva deste ABC, um *recurso* é um artefato binário que é enviado dentro de um pacote. Normalmente isso é algo como um arquivo de dados que fica próximo ao arquivo `__init__.py` do pacote. O objetivo desta classe é ajudar a abstrair o acesso a tais arquivos de dados para que não importe se o pacote e seu(s) arquivo(s) de dados estão armazenados em um arquivo, por exemplo, zip versus no sistema de arquivos.

Para qualquer um dos métodos desta classe, espera-se que um argumento *rescurso* seja um objeto tipo arquivo que representa conceitualmente apenas um nome de arquivo. Isso significa que nenhum caminho de subdiretório deve ser incluído no argumento *resource*. Isso ocorre porque a localização do pacote para o qual o leitor se destina, atua como o "diretório". Portanto, a metáfora para diretórios e nomes de arquivos são pacotes e recursos, respectivamente. É também por isso que se espera que as instâncias dessa classe se correlacionem diretamente a um pacote específico (em vez de representar potencialmente vários pacotes ou um módulo).

Carregadores que desejam oferecer suporte à leitura de recursos devem fornecer um método chamado `get_resource_reader(nomecompleto)` que retorna um objeto implementando esta interface ABC. Se o módulo especificado por nomecompleto não for um pacote, este método deve retornar *None*. Um objeto compatível com este ABC só deve ser retornado quando o módulo especificado for um pacote.

Novo na versão 3.7.

abstractmethod open_resource(*resource*)
Retorna um objeto tipo arquivo aberto para leitura binária de *resource*.

Se o recurso não puder ser encontrado, *FileNotFoundError* é levantada.

abstractmethod resource_path(*resource*)
Retorna o caminho do sistema de arquivos para *resource*.

Se o recurso não existir concretamente no sistema de arquivos, levanta *FileNotFoundError*.

abstractmethod is_resource(*name*)
Retorna `True` se o *name* nomeado for considerado um recurso. *FileNotFoundError* é levantada se *name* não existir.

abstractmethod contents()
Retorna um *iterável* de strings sobre o conteúdo do pacote. Observe que não é necessário que todos os nomes retornados pelo iterador sejam recursos reais, por exemplo, é aceitável retornar nomes para os quais *is_resource()* seria falso.

Permitir que nomes que não são recursos sejam retornados é permitir situações em que a forma como um pacote e seus recursos são armazenados é conhecida a priori e os nomes que não são recursos seriam úteis. Por exemplo, o retorno de nomes de subdiretórios é permitido para que, quando se souber que o pacote e os recursos estão armazenados no sistema de arquivos, esses nomes de subdiretórios possam ser usados diretamente.

O método abstrato retorna um iterável sem itens.

class importlib.abc.**ResourceLoader**
An abstract base class for a *loader* which implements the optional **PEP 302** protocol for loading arbitrary resources from the storage back-end.

Obsoleto desde a versão 3.7: This ABC is deprecated in favour of supporting resource loading through *importlib.abc.ResourceReader*.

abstractmethod get_data(*path*)
An abstract method to return the bytes for the data located at *path*. Loaders that have a file-like storage back-end that allows storing arbitrary data can implement this abstract method to give direct access to the data stored. *OSError* is to be raised if the *path* cannot be found. The *path* is expected to be constructed using a module's `__file__` attribute or an item from a package's `__path__`.

Alterado na versão 3.4: Raises *OSError* instead of *NotImplementedError*.

class importlib.abc.**InspectLoader**

An abstract base class for a *loader* which implements the optional **PEP 302** protocol for loaders that inspect modules.

get_code(*fullname*)

Return the code object for a module, or `None` if the module does not have a code object (as would be the case, for example, for a built-in module). Raise an `ImportError` if loader cannot find the requested module.

Nota: While the method has a default implementation, it is suggested that it be overridden if possible for performance.

Alterado na versão 3.4: No longer abstract and a concrete implementation is provided.

abstractmethod get_source(*fullname*)

An abstract method to return the source of a module. It is returned as a text string using *universal newlines*, translating all recognized line separators into `'\n'` characters. Returns `None` if no source is available (e.g. a built-in module). Raises `ImportError` if the loader cannot find the module specified.

Alterado na versão 3.4: Raises `ImportError` instead of `NotImplementedError`.

is_package(*fullname*)

An abstract method to return a true value if the module is a package, a false value otherwise. `ImportError` is raised if the *loader* cannot find the module.

Alterado na versão 3.4: Raises `ImportError` instead of `NotImplementedError`.

static source_to_code(*data*, *path='<string>'*)

Create a code object from Python source.

The *data* argument can be whatever the `compile()` function supports (i.e. string or bytes). The *path* argument should be the "path" to where the source code originated from, which can be an abstract concept (e.g. location in a zip file).

With the subsequent code object one can execute it in a module by running `exec(code, module.__dict__)`.

Novo na versão 3.4.

Alterado na versão 3.5: Made the method static.

exec_module(*module*)

Implementation of `Loader.exec_module()`.

Novo na versão 3.4.

load_module(*fullname*)

Implementation of `Loader.load_module()`.

Obsoleto desde a versão 3.4: use `exec_module()` instead.

class importlib.abc.**ExecutionLoader**

An abstract base class which inherits from `InspectLoader` that, when implemented, helps a module to be executed as a script. The ABC represents an optional **PEP 302** protocol.

abstractmethod get_filename(*fullname*)

An abstract method that is to return the value of `__file__` for the specified module. If no path is available, `ImportError` is raised.

If source code is available, then the method should return the path to the source file, regardless of whether a bytecode was used to load the module.

Alterado na versão 3.4: Raises `ImportError` instead of `NotImplementedError`.

class importlib.abc.**FileLoader**(*fullname*, *path*)
An abstract base class which inherits from *ResourceLoader* and *ExecutionLoader*, providing concrete implementations of *ResourceLoader.get_data()* and *ExecutionLoader.get_filename()*.

The *fullname* argument is a fully resolved name of the module the loader is to handle. The *path* argument is the path to the file for the module.

Novo na versão 3.3.

name
The name of the module the loader can handle.

path
Caminho para o arquivo do módulo

load_module(*fullname*)
Calls super's `load_module()`.

Obsoleto desde a versão 3.4: Use *Loader.exec_module()* instead.

abstractmethod get_filename(*fullname*)
Returns *path*.

abstractmethod get_data(*path*)
Reads *path* as a binary file and returns the bytes from it.

class importlib.abc.**SourceLoader**
An abstract base class for implementing source (and optionally bytecode) file loading. The class inherits from both *ResourceLoader* and *ExecutionLoader*, requiring the implementation of:

- *ResourceLoader.get_data()*
- ***ExecutionLoader.get_filename()*** Should only return the path to the source file; sourceless loading is not supported.

The abstract methods defined by this class are to add optional bytecode file support. Not implementing these optional methods (or causing them to raise *NotImplementedError*) causes the loader to only work with source code. Implementing the methods allows the loader to work with source *and* bytecode files; it does not allow for *sourceless* loading where only bytecode is provided. Bytecode files are an optimization to speed up loading by removing the parsing step of Python's compiler, and so no bytecode-specific API is exposed.

path_stats(*path*)
Optional abstract method which returns a *dict* containing metadata about the specified path. Supported dictionary keys are:

- `'mtime'` (mandatory): an integer or floating-point number representing the modification time of the source code;
- `'size'` (optional): the size in bytes of the source code.

Any other keys in the dictionary are ignored, to allow for future extensions. If the path cannot be handled, *OSError* is raised.

Novo na versão 3.3.

Alterado na versão 3.4: Raise *OSError* instead of *NotImplementedError*.

path_mtime(*path*)
Optional abstract method which returns the modification time for the specified path.

Obsoleto desde a versão 3.3: This method is deprecated in favour of *path_stats()*. You don't have to implement it, but it is still available for compatibility purposes. Raise *OSError* if the path cannot be handled.

Alterado na versão 3.4: Raise *OSError* instead of *NotImplementedError*.

set_data(*path*, *data*)
Optional abstract method which writes the specified bytes to a file path. Any intermediate directories which do not exist are to be created automatically.

When writing to the path fails because the path is read-only (*errno.EACCES*/*PermissionError*), do not propagate the exception.

Alterado na versão 3.4: No longer raises *NotImplementedError* when called.

get_code(*fullname*)
Concrete implementation of *InspectLoader.get_code()*.

exec_module(*module*)
Concrete implementation of *Loader.exec_module()*.

Novo na versão 3.4.

load_module(*fullname*)
Concrete implementation of *Loader.load_module()*.

Obsoleto desde a versão 3.4: Use *exec_module()* instead.

get_source(*fullname*)
Concrete implementation of *InspectLoader.get_source()*.

is_package(*fullname*)
Concrete implementation of *InspectLoader.is_package()*. A module is determined to be a package if its file path (as provided by *ExecutionLoader.get_filename()*) is a file named `__init__` when the file extension is removed **and** the module name itself does not end in `__init__`.

32.5.4 importlib.resources – Recursos

Source code: Lib/importlib/resources.py

Novo na versão 3.7.

This module leverages Python's import system to provide access to *resources* within *packages*. If you can import a package, you can access resources within that package. Resources can be opened or read, in either binary or text mode.

Resources are roughly akin to files inside directories, though it's important to keep in mind that this is just a metaphor. Resources and packages **do not** have to exist as physical files and directories on the file system.

Nota: This module provides functionality similar to pkg_resources Basic Resource Access without the performance overhead of that package. This makes reading resources included in packages easier, with more stable and consistent semantics.

The standalone backport of this module provides more information on using importlib.resources and migrating from pkg_resources to importlib.resources.

Loaders that wish to support resource reading should implement a `get_resource_reader(fullname)` method as specified by *importlib.abc.ResourceReader*.

The following types are defined.

importlib.resources.**Package**
The `Package` type is defined as `Union[str, ModuleType]`. This means that where the function describes accepting a `Package`, you can pass in either a string or a module. Module objects must have a resolvable `__spec__.submodule_search_locations` that is not `None`.

importlib.resources.**Resource**
This type describes the resource names passed into the various functions in this package. This is defined as `Union[str, os.PathLike]`.

The following functions are available.

importlib.resources.**open_binary**(*package*, *resource*)
Open for binary reading the *resource* within *package*.

package is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). This function returns a `typing.BinaryIO` instance, a binary I/O stream open for reading.

importlib.resources.**open_text**(*package*, *resource*, *encoding='utf-8'*, *errors='strict'*)
Open for text reading the *resource* within *package*. By default, the resource is opened for reading as UTF-8.

package is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). *encoding* and *errors* have the same meaning as with built-in *open()*.

This function returns a `typing.TextIO` instance, a text I/O stream open for reading.

importlib.resources.**read_binary**(*package*, *resource*)
Read and return the contents of the *resource* within *package* as `bytes`.

package is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). This function returns the contents of the resource as *bytes*.

importlib.resources.**read_text**(*package*, *resource*, *encoding='utf-8'*, *errors='strict'*)
Read and return the contents of *resource* within *package* as a `str`. By default, the contents are read as strict UTF-8.

package is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory). *encoding* and *errors* have the same meaning as with built-in *open()*. This function returns the contents of the resource as *str*.

importlib.resources.**path**(*package*, *resource*)
Return the path to the *resource* as an actual file system path. This function returns a context manager for use in a `with` statement. The context manager provides a *pathlib.Path* object.

Exiting the context manager cleans up any temporary file created when the resource needs to be extracted from e.g. a zip file.

package is either a name or a module object which conforms to the `Package` requirements. *resource* is the name of the resource to open within *package*; it may not contain path separators and it may not have sub-resources (i.e. it cannot be a directory).

importlib.resources.**is_resource**(*package*, *name*)
Return `True` if there is a resource named *name* in the package, otherwise `False`. Remember that directories are *not* resources! *package* is either a name or a module object which conforms to the `Package` requirements.

importlib.resources.**contents**(*package*)
Return an iterable over the named items within the package. The iterable returns *str* resources (e.g. files) and non-resources (e.g. directories). The iterable does not recurse into subdirectories.

package is either a name or a module object which conforms to the `Package` requirements.

32.5.5 `importlib.machinery` – Importers and path hooks

Source code: Lib/importlib/machinery.py

This module contains the various objects that help `import` find and load modules.

`importlib.machinery.`**`SOURCE_SUFFIXES`**
A list of strings representing the recognized file suffixes for source modules.

Novo na versão 3.3.

`importlib.machinery.`**`DEBUG_BYTECODE_SUFFIXES`**
A list of strings representing the file suffixes for non-optimized bytecode modules.

Novo na versão 3.3.

Obsoleto desde a versão 3.5: Use *`BYTECODE_SUFFIXES`* instead.

`importlib.machinery.`**`OPTIMIZED_BYTECODE_SUFFIXES`**
A list of strings representing the file suffixes for optimized bytecode modules.

Novo na versão 3.3.

Obsoleto desde a versão 3.5: Use *`BYTECODE_SUFFIXES`* instead.

`importlib.machinery.`**`BYTECODE_SUFFIXES`**
A list of strings representing the recognized file suffixes for bytecode modules (including the leading dot).

Novo na versão 3.3.

Alterado na versão 3.5: The value is no longer dependent on `__debug__`.

`importlib.machinery.`**`EXTENSION_SUFFIXES`**
A list of strings representing the recognized file suffixes for extension modules.

Novo na versão 3.3.

`importlib.machinery.`**`all_suffixes`**`()`
Returns a combined list of strings representing all file suffixes for modules recognized by the standard import machinery. This is a helper for code which simply needs to know if a filesystem path potentially refers to a module without needing any details on the kind of module (for example, *`inspect.getmodulename()`*).

Novo na versão 3.3.

class `importlib.machinery.`**`BuiltinImporter`**
An *importer* for built-in modules. All known built-in modules are listed in *`sys.builtin_module_names`*. This class implements the *`importlib.abc.MetaPathFinder`* and *`importlib.abc.InspectLoader`* ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

Alterado na versão 3.5: As part of **PEP 489**, the builtin importer now implements `Loader.create_module()` and `Loader.exec_module()`

class `importlib.machinery.`**`FrozenImporter`**
An *importer* for frozen modules. This class implements the *`importlib.abc.MetaPathFinder`* and *`importlib.abc.InspectLoader`* ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

Alterado na versão 3.4: Gained `create_module()` and `exec_module()` methods.

class importlib.machinery.**WindowsRegistryFinder**
Finder for modules declared in the Windows registry. This class implements the *importlib.abc.MetaPathFinder* ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

Novo na versão 3.3.

Obsoleto desde a versão 3.6: Use *site* configuration instead. Future versions of Python may not enable this finder by default.

class importlib.machinery.**PathFinder**
A *Finder* for *sys.path* and package __path__ attributes. This class implements the *importlib.abc.MetaPathFinder* ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

classmethod find_spec(*fullname*, *path=None*, *target=None*)
Class method that attempts to find a *spec* for the module specified by *fullname* on *sys.path* or, if defined, on *path*. For each path entry that is searched, *sys.path_importer_cache* is checked. If a non-false object is found then it is used as the *path entry finder* to look for the module being searched for. If no entry is found in *sys.path_importer_cache*, then *sys.path_hooks* is searched for a finder for the path entry and, if found, is stored in *sys.path_importer_cache* along with being queried about the module. If no finder is ever found then None is both stored in the cache and returned.

Novo na versão 3.4.

Alterado na versão 3.5: If the current working directory – represented by an empty string – is no longer valid then None is returned but no value is cached in *sys.path_importer_cache*.

classmethod find_module(*fullname*, *path=None*)
A legacy wrapper around *find_spec()*.

Obsoleto desde a versão 3.4: Use *find_spec()* instead.

classmethod invalidate_caches()
Calls *importlib.abc.PathEntryFinder.invalidate_caches()* on all finders stored in *sys.path_importer_cache* that define the method. Otherwise entries in *sys.path_importer_cache* set to None are deleted.

Alterado na versão 3.7: Entries of None in *sys.path_importer_cache* are deleted.

Alterado na versão 3.4: Calls objects in *sys.path_hooks* with the current working directory for '' (i.e. the empty string).

class importlib.machinery.**FileFinder**(*path*, **loader_details*)
A concrete implementation of *importlib.abc.PathEntryFinder* which caches results from the file system.

The *path* argument is the directory for which the finder is in charge of searching.

The *loader_details* argument is a variable number of 2-item tuples each containing a loader and a sequence of file suffixes the loader recognizes. The loaders are expected to be callables which accept two arguments of the module's name and the path to the file found.

The finder will cache the directory contents as necessary, making stat calls for each module search to verify the cache is not outdated. Because cache staleness relies upon the granularity of the operating system's state information of the file system, there is a potential race condition of searching for a module, creating a new file, and then searching for the module the new file represents. If the operations happen fast enough to fit within the granularity of stat calls, then the module search will fail. To prevent this from happening, when you create a module dynamically, make sure to call *importlib.invalidate_caches()*.

Novo na versão 3.3.

path
The path the finder will search in.

find_spec(*fullname*, *target=None*)
Attempt to find the spec to handle *fullname* within `path`.

Novo na versão 3.4.

find_loader(*fullname*)
Attempt to find the loader to handle *fullname* within `path`.

invalidate_caches()
Clear out the internal cache.

classmethod path_hook(**loader_details*)
A class method which returns a closure for use on `sys.path_hooks`. An instance of `FileFinder` is returned by the closure using the path argument given to the closure directly and *loader_details* indirectly.

If the argument to the closure is not an existing directory, `ImportError` is raised.

class importlib.machinery.**SourceFileLoader**(*fullname*, *path*)
A concrete implementation of `importlib.abc.SourceLoader` by subclassing `importlib.abc.FileLoader` and providing some concrete implementations of other methods.

Novo na versão 3.3.

name
The name of the module that this loader will handle.

path
The path to the source file.

is_package(*fullname*)
Return `True` if `path` appears to be for a package.

path_stats(*path*)
Concrete implementation of `importlib.abc.SourceLoader.path_stats()`.

set_data(*path*, *data*)
Concrete implementation of `importlib.abc.SourceLoader.set_data()`.

load_module(*name=None*)
Concrete implementation of `importlib.abc.Loader.load_module()` where specifying the name of the module to load is optional.

Obsoleto desde a versão 3.6: Use `importlib.abc.Loader.exec_module()` instead.

class importlib.machinery.**SourcelessFileLoader**(*fullname*, *path*)
A concrete implementation of `importlib.abc.FileLoader` which can import bytecode files (i.e. no source code files exist).

Please note that direct use of bytecode files (and thus not source code files) inhibits your modules from being usable by all Python implementations or new versions of Python which change the bytecode format.

Novo na versão 3.3.

name
The name of the module the loader will handle.

path
The path to the bytecode file.

is_package(*fullname*)
Determines if the module is a package based on `path`.

get_code(*fullname*)
Returns the code object for *name* created from *path*.

get_source(*fullname*)
Returns `None` as bytecode files have no source when this loader is used.

load_module(*name=None*)

Concrete implementation of *importlib.abc.Loader.load_module()* where specifying the name of the module to load is optional.

Obsoleto desde a versão 3.6: Use *importlib.abc.Loader.exec_module()* instead.

class importlib.machinery.**ExtensionFileLoader**(*fullname*, *path*)
A concrete implementation of *importlib.abc.ExecutionLoader* for extension modules.

The *fullname* argument specifies the name of the module the loader is to support. The *path* argument is the path to the extension module's file.

Novo na versão 3.3.

name
Name of the module the loader supports.

path
Path to the extension module.

create_module(*spec*)
Creates the module object from the given specification in accordance with **PEP 489**.

Novo na versão 3.5.

exec_module(*module*)
Initializes the given module object in accordance with **PEP 489**.

Novo na versão 3.5.

is_package(*fullname*)
Returns `True` if the file path points to a package's `__init__` module based on *EXTENSION_SUFFIXES*.

get_code(*fullname*)
Returns `None` as extension modules lack a code object.

get_source(*fullname*)
Returns `None` as extension modules do not have source code.

get_filename(*fullname*)
Returns *path*.

Novo na versão 3.4.

class importlib.machinery.**ModuleSpec**(*name*, *loader*, ***, *origin=None*, *loader_state=None*, *is_package=None*)
A specification for a module's import-system-related state. This is typically exposed as the module's `__spec__` attribute. In the descriptions below, the names in parentheses give the corresponding attribute available directly on the module object. E.g. `module.__spec__.origin == module.__file__`. Note however that while the *values* are usually equivalent, they can differ since there is no synchronization between the two objects. Thus it is possible to update the module's `__path__` at runtime, and this will not be automatically reflected in `__spec__.submodule_search_locations`.

Novo na versão 3.4.

name

(__name__)

A string for the fully-qualified name of the module.

loader

(__loader__)

The loader to use for loading. For namespace packages this should be set to None.

origin

(__file__)

Name of the place from which the module is loaded, e.g. "builtin" for built-in modules and the filename for modules loaded from source. Normally "origin" should be set, but it may be None (the default) which indicates it is unspecified (e.g. for namespace packages).

submodule_search_locations

(__path__)

List of strings for where to find submodules, if a package (None otherwise).

loader_state

Container of extra module-specific data for use during loading (or None).

cached

(__cached__)

String for where the compiled module should be stored (or None).

parent

`(``__package__)"`

(Read-only) Fully-qualified name of the package to which the module belongs as a submodule (or None).

has_location

Boolean indicating whether or not the module's "origin" attribute refers to a loadable location.

32.5.6 importlib.util – Utility code for importers

Source code: Lib/importlib/util.py

This module contains the various objects that help in the construction of an *importer*.

importlib.util.**MAGIC_NUMBER**
The bytes which represent the bytecode version number. If you need help with loading/writing bytecode then consider *importlib.abc.SourceLoader*.

Novo na versão 3.4.

importlib.util.**cache_from_source**(*path*, *debug_override=None*, *, *optimization=None*)
Return the **PEP 3147**/**PEP 488** path to the byte-compiled file associated with the source *path*. For example, if *path* is /foo/bar/baz.py the return value would be /foo/bar/__pycache__/baz.cpython-32.pyc for Python 3.2. The cpython-32 string comes from the current magic tag (see get_tag(); if sys.implementation.cache_tag is not defined then *NotImplementedError* will be raised).

The *optimization* parameter is used to specify the optimization level of the bytecode file. An empty string represents no optimization, so `/foo/bar/baz.py` with an *optimization* of `''` will result in a bytecode path of `/foo/bar/__pycache__/baz.cpython-32.pyc`. `None` causes the interpter's optimization level to be used. Any other value's string representation being used, so `/foo/bar/baz.py` with an *optimization* of `2` will lead to the bytecode path of `/foo/bar/__pycache__/baz.cpython-32.opt-2.pyc`. The string representation of *optimization* can only be alphanumeric, else *ValueError* is raised.

The *debug_override* parameter is deprecated and can be used to override the system's value for `__debug__`. A `True` value is the equivalent of setting *optimization* to the empty string. A `False` value is the same as setting *optimization* to `1`. If both *debug_override* an *optimization* are not `None` then *TypeError* is raised.

Novo na versão 3.4.

Alterado na versão 3.5: The *optimization* parameter was added and the *debug_override* parameter was deprecated.

Alterado na versão 3.6: Aceita um *path-like object*.

`importlib.util.`**`source_from_cache`**(*path*)

Given the *path* to a **PEP 3147** file name, return the associated source code file path. For example, if *path* is `/foo/bar/__pycache__/baz.cpython-32.pyc` the returned path would be `/foo/bar/baz.py`. *path* need not exist, however if it does not conform to **PEP 3147** or **PEP 488** format, a *ValueError* is raised. If `sys.implementation.cache_tag` is not defined, *NotImplementedError* is raised.

Novo na versão 3.4.

Alterado na versão 3.6: Aceita um *path-like object*.

`importlib.util.`**`decode_source`**(*source_bytes*)

Decode the given bytes representing source code and return it as a string with universal newlines (as required by *importlib.abc.InspectLoader.get_source()*).

Novo na versão 3.4.

`importlib.util.`**`resolve_name`**(*name*, *package*)

Resolve a relative module name to an absolute one.

If **name** has no leading dots, then **name** is simply returned. This allows for usage such as `importlib.util.resolve_name('sys', __package__)` without doing a check to see if the **package** argument is needed.

ValueError is raised if **name** is a relative module name but package is a false value (e.g. `None` or the empty string). *ValueError* is also raised a relative name would escape its containing package (e.g. requesting `..bacon` from within the `spam` package).

Novo na versão 3.3.

`importlib.util.`**`find_spec`**(*name*, *package=None*)

Find the *spec* for a module, optionally relative to the specified **package** name. If the module is in *sys.modules*, then `sys.modules[name].__spec__` is returned (unless the spec would be `None` or is not set, in which case *ValueError* is raised). Otherwise a search using *sys.meta_path* is done. `None` is returned if no spec is found.

If **name** is for a submodule (contains a dot), the parent module is automatically imported.

name and **package** work the same as for `import_module()`.

Novo na versão 3.4.

Alterado na versão 3.7: Raises *ModuleNotFoundError* instead of *AttributeError* if **package** is in fact not a package (i.e. lacks a `__path__` attribute).

`importlib.util.`**`module_from_spec`**(*spec*)

Create a new module based on **spec** and *spec.loader.create_module*.

If *spec.loader.create_module* does not return None, then any pre-existing attributes will not be reset. Also, no *AttributeError* will be raised if triggered while accessing **spec** or setting an attribute on the module.

This function is preferred over using *types.ModuleType* to create a new module as **spec** is used to set as many import-controlled attributes on the module as possible.

Novo na versão 3.5.

@importlib.util.**module_for_loader**

A *decorator* for *importlib.abc.Loader.load_module()* to handle selecting the proper module object to load with. The decorated method is expected to have a call signature taking two positional arguments (e.g. load_module(self, module)) for which the second argument will be the module **object** to be used by the loader. Note that the decorator will not work on static methods because of the assumption of two arguments.

The decorated method will take in the **name** of the module to be loaded as expected for a *loader*. If the module is not found in *sys.modules* then a new one is constructed. Regardless of where the module came from, __loader__ set to **self** and __package__ is set based on what *importlib.abc.InspectLoader.is_package()* returns (if available). These attributes are set unconditionally to support reloading.

If an exception is raised by the decorated method and a module was added to *sys.modules*, then the module will be removed to prevent a partially initialized module from being in left in *sys.modules*. If the module was already in *sys.modules* then it is left alone.

Alterado na versão 3.3: __loader__ and __package__ are automatically set (when possible).

Alterado na versão 3.4: Set __name__, __loader__ __package__ unconditionally to support reloading.

Obsoleto desde a versão 3.4: The import machinery now directly performs all the functionality provided by this function.

@importlib.util.**set_loader**

A *decorator* for *importlib.abc.Loader.load_module()* to set the __loader__ attribute on the returned module. If the attribute is already set the decorator does nothing. It is assumed that the first positional argument to the wrapped method (i.e. self) is what __loader__ should be set to.

Alterado na versão 3.4: Set __loader__ if set to None, as if the attribute does not exist.

Obsoleto desde a versão 3.4: The import machinery takes care of this automatically.

@importlib.util.**set_package**

A *decorator* for *importlib.abc.Loader.load_module()* to set the __package__ attribute on the returned module. If __package__ is set and has a value other than None it will not be changed.

Obsoleto desde a versão 3.4: The import machinery takes care of this automatically.

importlib.util.**spec_from_loader**(*name*, *loader*, ***, *origin=None*, *is_package=None*)

A factory function for creating a ModuleSpec instance based on a loader. The parameters have the same meaning as they do for ModuleSpec. The function uses available *loader* APIs, such as InspectLoader.is_package(), to fill in any missing information on the spec.

Novo na versão 3.4.

importlib.util.**spec_from_file_location**(*name*, *location*, ***, *loader=None*, *submodule_search_locations=None*)

A factory function for creating a ModuleSpec instance based on the path to a file. Missing information will be filled in on the spec by making use of loader APIs and by the implication that the module will be file-based.

Novo na versão 3.4.

Alterado na versão 3.6: Aceita um *path-like object*.

importlib.util.**source_hash**(*source_bytes*)

Return the hash of *source_bytes* as bytes. A hash-based .pyc file embeds the *source_hash()* of the corresponding source file's contents in its header.

Novo na versão 3.7.

class importlib.util.**LazyLoader**(*loader*)

A class which postpones the execution of the loader of a module until the module has an attribute accessed.

This class **only** works with loaders that define *exec_module()* as control over what module type is used for the module is required. For those same reasons, the loader's *create_module()* method must return None or a type for which its __class__ attribute can be mutated along with not using *slots*. Finally, modules which substitute the object placed into *sys.modules* will not work as there is no way to properly replace the module references throughout the interpreter safely; *ValueError* is raised if such a substitution is detected.

Nota: For projects where startup time is critical, this class allows for potentially minimizing the cost of loading a module if it is never used. For projects where startup time is not essential then use of this class is **heavily** discouraged due to error messages created during loading being postponed and thus occurring out of context.

Novo na versão 3.5.

Alterado na versão 3.6: Began calling *create_module()*, removing the compatibility warning for *importlib.machinery.BuiltinImporter* and *importlib.machinery.ExtensionFileLoader*.

classmethod factory(*loader*)

A static method which returns a callable that creates a lazy loader. This is meant to be used in situations where the loader is passed by class instead of by instance.

```
suffixes = importlib.machinery.SOURCE_SUFFIXES
loader = importlib.machinery.SourceFileLoader
lazy_loader = importlib.util.LazyLoader.factory(loader)
finder = importlib.machinery.FileFinder(path, (lazy_loader, suffixes))
```

32.5.7 Exemplos

Importing programmatically

To programmatically import a module, use *importlib.import_module()*.

```
import importlib

itertools = importlib.import_module('itertools')
```

Checando se o módulo pode ser importado

If you need to find out if a module can be imported without actually doing the import, then you should use *importlib.util.find_spec()*.

```
import importlib.util
import sys

# For illustrative purposes.
name = 'itertools'

spec = importlib.util.find_spec(name)
if spec is None:
```

(continua na próxima página)

(continuação da página anterior)

```
    print("can't find the itertools module")
else:
    # If you chose to perform the actual import ...
    module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(module)
    # Adding the module to sys.modules is optional.
    sys.modules[name] = module
```

Importa o arquivo de origem diretamente

Importação um arquivo Python diretamente da fonte, use o seguinte caminho (Somente versões Python 3.5 acima)

```
import importlib.util
import sys

# For illustrative purposes.
import tokenize
file_path = tokenize.__file__
module_name = tokenize.__name__

spec = importlib.util.spec_from_file_location(module_name, file_path)
module = importlib.util.module_from_spec(spec)
spec.loader.exec_module(module)
# Optional; only necessary if you want to be able to import the module
# by name later.
sys.modules[module_name] = module
```

Setting up an importer

For deep customizations of import, you typically want to implement an *importer*. This means managing both the *finder* and *loader* side of things. For finders there are two flavours to choose from depending on your needs: a *meta path finder* or a *path entry finder*. The former is what you would put on `sys.meta_path` while the latter is what you create using a *path entry hook* on `sys.path_hooks` which works with `sys.path` entries to potentially create a finder. This example will show you how to register your own importers so that import will use them (for creating an importer for yourself, read the documentation for the appropriate classes defined within this package):

```
import importlib.machinery
import sys

# For illustrative purposes only.
SpamMetaPathFinder = importlib.machinery.PathFinder
SpamPathEntryFinder = importlib.machinery.FileFinder
loader_details = (importlib.machinery.SourceFileLoader,
                  importlib.machinery.SOURCE_SUFFIXES)

# Setting up a meta path finder.
# Make sure to put the finder in the proper location in the list in terms of
# priority.
sys.meta_path.append(SpamMetaPathFinder)

# Setting up a path entry finder.
# Make sure to put the path hook in the proper location in the list in terms
```

(continua na próxima página)

(continuação da página anterior)

```
# of priority.
sys.path_hooks.append(SpamPathEntryFinder.path_hook(loader_details))
```

Approximating `importlib.import_module()`

Import itself is implemented in Python code, making it possible to expose most of the import machinery through importlib. The following helps illustrate the various APIs that importlib exposes by providing an approximate implementation of *importlib.import_module()* (Python 3.4 and newer for the importlib usage, Python 3.6 and newer for other parts of the code).

```
import importlib.util
import sys

def import_module(name, package=None):
    """An approximate implementation of import."""
    absolute_name = importlib.util.resolve_name(name, package)
    try:
        return sys.modules[absolute_name]
    except KeyError:
        pass

    path = None
    if '.' in absolute_name:
        parent_name, _, child_name = absolute_name.rpartition('.')
        parent_module = import_module(parent_name)
        path = parent_module.__spec__.submodule_search_locations
    for finder in sys.meta_path:
        spec = finder.find_spec(absolute_name, path)
        if spec is not None:
            break
    else:
        msg = f'No module named {absolute_name!r}'
        raise ModuleNotFoundError(msg, name=absolute_name)
    module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(module)
    sys.modules[absolute_name] = module
    if path is not None:
        setattr(parent_module, child_name, module)
    return module
```

CAPÍTULO 33

Serviços da Linguagem Python

O Python fornece vários módulos para ajudar a trabalhar com a linguagem Python. Estes módulos suportam tokenizing, parsing, análise de sintaxe, disassembly de bytecode e várias outras facilidades.

Esses módulos incluem:

33.1 `parser` — Acessa árvores de análise do Python

O módulo *`parser`* fornece uma interface para o analisador sintático interno do Python e o compilador de código de bytes. O objetivo principal dessa interface é permitir que o código Python edite a árvore de análise de uma expressão Python e crie código executável a partir disso. Isso é melhor do que tentar analisar e modificar um fragmento de código Python arbitrário como uma string, porque a análise é realizada de maneira idêntica ao código que forma o aplicativo. Também é mais rápido.

Nota: A partir do Python 2.5 em diante, é muito mais conveniente interromper o estágio de geração e compilação da Árvore de Sintaxe Abstrata (AST), usando o módulo: mod:*ast*.

Há algumas coisas a serem observadas sobre este módulo que são importantes para fazer uso das estruturas de dados criadas. Este não é um tutorial sobre como editar as árvores de análise para o código Python, mas são apresentados alguns exemplos de uso do módulo *`parser`*.

Most importantly, a good understanding of the Python grammar processed by the internal parser is required. For full information on the language syntax, refer to reference-index. The parser itself is created from a grammar specification defined in the file `Grammar/Grammar` in the standard Python distribution. The parse trees stored in the ST objects created by this module are the actual output from the internal parser when created by the *`expr()`* or *`suite()`* functions, described below. The ST objects created by *`sequence2st()`* faithfully simulate those structures. Be aware that the values of the sequences which are considered "correct" will vary from one version of Python to another as the formal grammar for the language is revised. However, transporting code from one Python version to another as source text will always allow correct parse trees to be created in the target version, with the only restriction being that migrating to an older

version of the interpreter will not support more recent language constructs. The parse trees are not typically compatible from one version to another, though source code has usually been forward-compatible within a major release series.

Each element of the sequences returned by *st2list()* or *st2tuple()* has a simple form. Sequences representing non-terminal elements in the grammar always have a length greater than one. The first element is an integer which identifies a production in the grammar. These integers are given symbolic names in the C header file `Include/graminit.h` and the Python module *symbol*. Each additional element of the sequence represents a component of the production as recognized in the input string: these are always sequences which have the same form as the parent. An important aspect of this structure which should be noted is that keywords used to identify the parent node type, such as the keyword `if` in an `if_stmt`, are included in the node tree without any special treatment. For example, the `if` keyword is represented by the tuple `(1, 'if')`, where `1` is the numeric value associated with all `NAME` tokens, including variable and function names defined by the user. In an alternate form returned when line number information is requested, the same token might be represented as `(1, 'if', 12)`, where the `12` represents the line number at which the terminal symbol was found.

Terminal elements are represented in much the same way, but without any child elements and the addition of the source text which was identified. The example of the `if` keyword above is representative. The various types of terminal symbols are defined in the C header file `Include/token.h` and the Python module *token*.

The ST objects are not required to support the functionality of this module, but are provided for three purposes: to allow an application to amortize the cost of processing complex parse trees, to provide a parse tree representation which conserves memory space when compared to the Python list or tuple representation, and to ease the creation of additional modules in C which manipulate parse trees. A simple "wrapper" class may be created in Python to hide the use of ST objects.

The *parser* module defines functions for a few distinct purposes. The most important purposes are to create ST objects and to convert ST objects to other representations such as parse trees and compiled code objects, but there are also functions which serve to query the type of parse tree represented by an ST object.

Ver também:

Módulo *symbol* Useful constants representing internal nodes of the parse tree.

Módulo *token* Useful constants representing leaf nodes of the parse tree and functions for testing node values.

33.1.1 Creating ST Objects

ST objects may be created from source code or from a parse tree. When creating an ST object from source, different functions are used to create the `'eval'` and `'exec'` forms.

`parser.`**`expr`**(*source*)
The *expr()* function parses the parameter *source* as if it were an input to `compile(source, 'file.py', 'eval')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

`parser.`**`suite`**(*source*)
The *suite()* function parses the parameter *source* as if it were an input to `compile(source, 'file.py', 'exec')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

`parser.`**`sequence2st`**(*sequence*)
This function accepts a parse tree represented as a sequence and builds an internal representation if possible. If it can validate that the tree conforms to the Python grammar and all nodes are valid node types in the host version of Python, an ST object is created from the internal representation and returned to the called. If there is a problem creating the internal representation, or if the tree cannot be validated, a *ParserError* exception is raised. An ST object created this way should not be assumed to compile correctly; normal exceptions raised by compilation may still be initiated when the ST object is passed to *compilest()*. This may indicate problems not related to syntax (such as a *MemoryError* exception), but may also be due to constructs such as the result of parsing `del f(0)`, which escapes the Python parser but is checked by the bytecode compiler.

Sequences representing terminal tokens may be represented as either two-element lists of the form `(1, 'name')` or as three-element lists of the form `(1, 'name', 56)`. If the third element is present, it is assumed to be a valid line number. The line number may be specified for any subset of the terminal symbols in the input tree.

`parser.`**`tuple2st`**(*sequence*)
This is the same function as *`sequence2st()`*. This entry point is maintained for backward compatibility.

33.1.2 Convertendo objetos ST

ST objects, regardless of the input used to create them, may be converted to parse trees represented as list- or tuple-trees, or may be compiled into executable code objects. Parse trees may be extracted with or without line numbering information.

`parser.`**`st2list`**(*st*, *line_info=False*, *col_info=False*)
This function accepts an ST object from the caller in *st* and returns a Python list representing the equivalent parse tree. The resulting list representation can be used for inspection or the creation of a new parse tree in list form. This function does not fail so long as memory is available to build the list representation. If the parse tree will only be used for inspection, *`st2tuple()`* should be used instead to reduce memory consumption and fragmentation. When the list representation is required, this function is significantly faster than retrieving a tuple representation and converting that to nested lists.

If *line_info* is true, line number information will be included for all terminal tokens as a third element of the list representing the token. Note that the line number provided specifies the line on which the token *ends*. This information is omitted if the flag is false or omitted.

`parser.`**`st2tuple`**(*st*, *line_info=False*, *col_info=False*)
This function accepts an ST object from the caller in *st* and returns a Python tuple representing the equivalent parse tree. Other than returning a tuple instead of a list, this function is identical to *`st2list()`*.

If *line_info* is true, line number information will be included for all terminal tokens as a third element of the list representing the token. This information is omitted if the flag is false or omitted.

`parser.`**`compilest`**(*st*, *filename='<syntax-tree>'*)
The Python byte compiler can be invoked on an ST object to produce code objects which can be used as part of a call to the built-in *`exec()`* or *`eval()`* functions. This function provides the interface to the compiler, passing the internal parse tree from *st* to the parser, using the source file name specified by the *filename* parameter. The default value supplied for *filename* indicates that the source was an ST object.

Compiling an ST object may result in exceptions related to compilation; an example would be a *`SyntaxError`* caused by the parse tree for `del f(0)`: this statement is considered legal within the formal grammar for Python but is not a legal language construct. The *`SyntaxError`* raised for this condition is actually generated by the Python byte-compiler normally, which is why it can be raised at this point by the *`parser`* module. Most causes of compilation failure can be diagnosed programmatically by inspection of the parse tree.

33.1.3 Queries on ST Objects

Two functions are provided which allow an application to determine if an ST was created as an expression or a suite. Neither of these functions can be used to determine if an ST was created from source code via *`expr()`* or *`suite()`* or from a parse tree via *`sequence2st()`*.

`parser.`**`isexpr`**(*st*)
When *st* represents an `'eval'` form, this function returns `True`, otherwise it returns `False`. This is useful, since code objects normally cannot be queried for this information using existing built-in functions. Note that the code objects created by *`compilest()`* cannot be queried like this either, and are identical to those created by the built-in *`compile()`* function.

parser.**issuite**(*st*)
: This function mirrors *isexpr()* in that it reports whether an ST object represents an `'exec'` form, commonly known as a "suite." It is not safe to assume that this function is equivalent to `not isexpr(st)`, as additional syntactic fragments may be supported in the future.

33.1.4 Exceptions and Error Handling

The parser module defines a single exception, but may also pass other built-in exceptions from other portions of the Python runtime environment. See each function for information about the exceptions it can raise.

exception parser.**ParserError**
: Exception raised when a failure occurs within the parser module. This is generally produced for validation failures rather than the built-in *SyntaxError* raised during normal parsing. The exception argument is either a string describing the reason of the failure or a tuple containing a sequence causing the failure from a parse tree passed to *sequence2st()* and an explanatory string. Calls to *sequence2st()* need to be able to handle either type of exception, while calls to other functions in the module will only need to be aware of the simple string values.

Note that the functions *compilest()*, *expr()*, and *suite()* may raise exceptions which are normally raised by the parsing and compilation process. These include the built in exceptions *MemoryError*, *OverflowError*, *SyntaxError*, and *SystemError*. In these cases, these exceptions carry all the meaning normally associated with them. Refer to the descriptions of each function for detailed information.

33.1.5 ST Objects

Ordered and equality comparisons are supported between ST objects. Pickling of ST objects (using the *pickle* module) is also supported.

parser.**STType**
: The type of the objects returned by *expr()*, *suite()* and *sequence2st()*.

ST objects have the following methods:

ST.**compile**(*filename='<syntax-tree>'*)
: Same as `compilest(st, filename)`.

ST.**isexpr**()
: Same as `isexpr(st)`.

ST.**issuite**()
: O mesmo que `issuite(st)`.

ST.**tolist**(*line_info=False*, *col_info=False*)
: Same as `st2list(st, line_info, col_info)`.

ST.**totuple**(*line_info=False*, *col_info=False*)
: Same as `st2tuple(st, line_info, col_info)`.

33.1.6 Example: Emulation of `compile()`

While many useful operations may take place between parsing and bytecode generation, the simplest operation is to do nothing. For this purpose, using the *parser* module to produce an intermediate data structure is equivalent to the code

```
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

The equivalent operation using the *parser* module is somewhat longer, and allows the intermediate internal parse tree to be retained as an ST object:

```
>>> import parser
>>> st = parser.expr('a + 5')
>>> code = st.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

An application which needs both ST and code objects can package this code into readily available functions:

```
import parser

def load_suite(source_string):
    st = parser.suite(source_string)
    return st, st.compile()

def load_expression(source_string):
    st = parser.expr(source_string)
    return st, st.compile()
```

33.2 `ast` — Árvores de Sintaxe Abstrata

Código-fonte: Lib/ast.py

O módulo *ast* ajuda os aplicativos Python a processar árvores da gramática de sintaxe abstrata do Python. A sintaxe abstrata em si pode mudar em cada lançamento do Python; este módulo ajuda a descobrir programaticamente como é a gramática atual.

Uma árvore de sintaxe abstrata pode ser gerada passando `ast.PyCF_ONLY_AST` como um sinalizador para a função interna *compile()*, ou usando o auxiliar *parse()* fornecido neste módulo. O resultado será uma árvore de objetos cujas classes herdam de *ast.AST*. Uma árvore de sintaxe abstrata pode ser compilada em um objeto de código Python usando a função interna *compile()*.

33.2.1 Classes de nó

class ast.**AST**

Esta é a base de todas as classes de nós AST. As classes de nós reais são derivadas do arquivo `Parser/Python.asdl`, o qual é reproduzido *abaixo*. Elas são definidas no módulo C `_ast` e reexportadas em *ast*.

Há uma classe definida para cada símbolo do lado esquerdo na gramática abstrata (por exemplo, `ast.stmt` ou `ast.expr`). Além disso, existe uma classe definida para cada construtor no lado direito; essas classes herdam das classes para as árvores do lado esquerdo. Por exemplo, `ast.BinOp` herda de `ast.expr`. Para regras de produção com alternativas ("somas"), a classe do lado esquerdo é abstrata: apenas instâncias de nós construtores específicos são criadas.

_fields

Cada classe concreta possui um atributo *_fields* que fornece os nomes de todos os nós filhos.

Cada instância de uma classe concreta tem um atributo para cada nó filho, do tipo definido na gramática. Por exemplo, as instâncias `ast.BinOp` possuem um atributo `left` do tipo `ast.expr`.

Se estes atributos estiverem marcados como opcionais na gramática (usando um ponto de interrogação), o valor pode ser `None`. Se os atributos puderem ter valor zero ou mais (marcados com um asterisco), os valores serão representados como listas do Python. Todos os atributos possíveis devem estar presentes e ter valores válidos ao compilar uma AST com *compile()*.

lineno
col_offset

Instâncias das subclasses `ast.expr` e `ast.stmt` possuem atributos *lineno* e *col_offset*. O *lineno* é o número da linha do texto fonte (indexado em 1, de modo que a primeira linha é a linha 1) e o *col_offset* é o deslocamento de bytes UTF-8 do primeiro token que gerou o nó. O deslocamento UTF-8 é registrado porque o analisador usa o UTF-8 internamente.

O construtor de uma classe `ast.T` analisa seus argumentos da seguinte forma:

- Se houver argumentos posicionais, deve haver tantos quanto houver itens em `T._fields`; eles serão atribuídos como atributos desses nomes.
- Se houver argumentos de palavra-chave, eles definirão os atributos dos mesmos nomes para os valores fornecidos.

Por exemplo, para criar e popular um nó `ast.UnaryOp`, você poderia usar

```
node = ast.UnaryOp()
node.op = ast.USub()
node.operand = ast.Num()
node.operand.n = 5
node.operand.lineno = 0
node.operand.col_offset = 0
node.lineno = 0
node.col_offset = 0
```

ou a forma mais compacta

```
node = ast.UnaryOp(ast.USub(), ast.Num(5, lineno=0, col_offset=0),
                   lineno=0, col_offset=0)
```

33.2.2 Gramática Abstrata

A gramática abstrata está atualmente definida da seguinte forma:

```
-- ASDL's 7 builtin types are:
-- identifier, int, string, bytes, object, singleton, constant
--
-- singleton: None, True or False
-- constant can be None, whereas None means "no value" for object.

module Python
{
    mod = Module(stmt* body)
        | Interactive(stmt* body)
        | Expression(expr body)

        -- not really an actual node but useful in Jython's typesystem.
        | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns)
          | AsyncFunctionDef(identifier name, arguments args,
                             stmt* body, expr* decorator_list, expr? returns)

          | ClassDef(identifier name,
             expr* bases,
             keyword* keywords,
             stmt* body,
             expr* decorator_list)
          | Return(expr? value)

          | Delete(expr* targets)
          | Assign(expr* targets, expr value)
          | AugAssign(expr target, operator op, expr value)
          -- 'simple' indicates that we annotate simple name without parens
          | AnnAssign(expr target, expr annotation, expr? value, int simple)

          -- use 'orelse' because else is a keyword in target languages
          | For(expr target, expr iter, stmt* body, stmt* orelse)
          | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse)
          | While(expr test, stmt* body, stmt* orelse)
          | If(expr test, stmt* body, stmt* orelse)
          | With(withitem* items, stmt* body)
          | AsyncWith(withitem* items, stmt* body)

          | Raise(expr? exc, expr? cause)
          | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
          | Assert(expr test, expr? msg)

          | Import(alias* names)
          | ImportFrom(identifier? module, alias* names, int? level)

          | Global(identifier* names)
          | Nonlocal(identifier* names)
          | Expr(expr value)
          | Pass | Break | Continue

          -- XXX Jython will be different
```

(continua na próxima página)

(continuação da página anterior)

```
         -- col_offset is the byte offset in the utf8 string the parser uses
         attributes (int lineno, int col_offset)

         -- BoolOp() can use left & right?
    expr = BoolOp(boolop op, expr* values)
         | BinOp(expr left, operator op, expr right)
         | UnaryOp(unaryop op, expr operand)
         | Lambda(arguments args, expr body)
         | IfExp(expr test, expr body, expr orelse)
         | Dict(expr* keys, expr* values)
         | Set(expr* elts)
         | ListComp(expr elt, comprehension* generators)
         | SetComp(expr elt, comprehension* generators)
         | DictComp(expr key, expr value, comprehension* generators)
         | GeneratorExp(expr elt, comprehension* generators)
         -- the grammar constrains where yield expressions can occur
         | Await(expr value)
         | Yield(expr? value)
         | YieldFrom(expr value)
         -- need sequences for compare to distinguish between
         -- x < 4 < 3 and (x < 4) < 3
         | Compare(expr left, cmpop* ops, expr* comparators)
         | Call(expr func, expr* args, keyword* keywords)
         | Num(object n) -- a number as a PyObject.
         | Str(string s) -- need to specify raw, unicode, etc?
         | FormattedValue(expr value, int? conversion, expr? format_spec)
         | JoinedStr(expr* values)
         | Bytes(bytes s)
         | NameConstant(singleton value)
         | Ellipsis
         | Constant(constant value)

         -- the following expression can appear in assignment context
         | Attribute(expr value, identifier attr, expr_context ctx)
         | Subscript(expr value, slice slice, expr_context ctx)
         | Starred(expr value, expr_context ctx)
         | Name(identifier id, expr_context ctx)
         | List(expr* elts, expr_context ctx)
         | Tuple(expr* elts, expr_context ctx)

         -- col_offset is the byte offset in the utf8 string the parser uses
         attributes (int lineno, int col_offset)

    expr_context = Load | Store | Del | AugLoad | AugStore | Param

    slice = Slice(expr? lower, expr? upper, expr? step)
          | ExtSlice(slice* dims)
          | Index(expr value)

    boolop = And | Or

    operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
                 | RShift | BitOr | BitXor | BitAnd | FloorDiv

    unaryop = Invert | Not | UAdd | USub

    cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn
```

(continua na próxima página)

(continuação da página anterior)

```
    comprehension = (expr target, expr iter, expr* ifs, int is_async)

    excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                    attributes (int lineno, int col_offset)

    arguments = (arg* args, arg? vararg, arg* kwonlyargs, expr* kw_defaults,
                 arg? kwarg, expr* defaults)

    arg = (identifier arg, expr? annotation)
           attributes (int lineno, int col_offset)

    -- keyword arguments supplied to call (NULL identifier for **kwargs)
    keyword = (identifier? arg, expr value)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)

    withitem = (expr context_expr, expr? optional_vars)
}
```

33.2.3 Auxiliares de `ast`

Além das classes de nós, o módulo `ast` define essas funções e classes utilitárias para percorrer árvores de sintaxe abstrata:

`ast.`**`parse`**(*source*, *filename='<unknown>'*, *mode='exec'*)
Analisa a fonte em um nó AST. Equivalente a `compile(source, filename, mode, ast.PyCF_ONLY_AST)`.

> **Aviso:** É possível travar o interpretador Python com uma string suficientemente grande/complexa devido às limitações de profundidade da pilha no compilador de AST do Python.

`ast.`**`literal_eval`**(*node_or_string*)
Avalia com segurança um nó de expressão ou uma string contendo um literal Python ou exibição de contêiner. A string ou o nó fornecido pode consistir apenas nas seguintes estruturas literais de Python: strings, bytes, números, tuplas, listas, dicts, conjuntos, booleanos e `None`.

Isso pode ser usado para avaliar com segurança strings contendo valores Python de fontes não confiáveis sem a necessidade de analisar os valores por si próprio. Não é capaz de avaliar expressões arbitrariamente complexas, por exemplo, envolvendo operadores ou indexação.

> **Aviso:** É possível travar o interpretador Python com uma string suficientemente grande/complexa devido às limitações de profundidade da pilha no compilador de AST do Python.

Alterado na versão 3.2: Agora permite bytes e literais de conjuntos.

`ast.`**`get_docstring`**(*node*, *clean=True*)
Retorna a docstring do *nó* dado (que deve ser um nó `FunctionDef`, `AsyncFunctionDef`, `ClassDef`

ou `Module`) ou `None` se não tiver uma docstring. Se *clean* for verdadeiro, limpa o recuo da docstring com *inspect.cleandoc()*.

Alterado na versão 3.5: Não há suporte a `AsyncFunctionDef`.

`ast.`**fix_missing_locations**(*node*)
Quando você compila uma árvore de nós com *compile()*, o compilador espera atributos :attr:‘ lineno‘ e `col_offset` para cada nó que os suporta. Isso é tedioso para preencher nós gerados, portanto, esse auxiliar adiciona esses atributos recursivamente, onde ainda não estão definidos, definindo-os para os valores do nó pai. Ele funciona recursivamente a partir do *node*.

`ast.`**increment_lineno**(*node*, *n=1*)
Incrementa o número da linha de cada nó na árvore, começando em *node* por *n*. Isso é útil para “mover código” para um local diferente em um arquivo.

`ast.`**copy_location**(*new_node*, *old_node*)
Copia o local de origem (`lineno` e `col_offset`) de *old_node* para *new_node* se possível, e retorna *new_node*.

`ast.`**iter_fields**(*node*)
Produz uma tupla de `(fieldname, value)` para cada campo em `node._fields` que esteja presente em *node*.

`ast.`**iter_child_nodes**(*node*)
Produz todos os nós filhos diretos de *node*, ou seja, todos os campos que são nós e todos os itens de campos que são listas de nós.

`ast.`**walk**(*node*)
Produz recursivamente todos os nós descendentes na árvore começando em *node* (incluindo o próprio *node*), em nenhuma ordem especificada. Isso é útil se você quiser apenas modificar nós no lugar e não se importar com o contexto.

class `ast.`**NodeVisitor**
Uma classe base de visitante de nó que percorre a árvore de sintaxe abstrata e chama uma função de visitante para cada nó encontrado. Esta função pode retornar um valor que é encaminhado pelo método *visit()*.

Esta classe deve ser uma subclasse, com a subclasse adicionando métodos visitantes.

visit(*node*)
Visita um nó. A implementação padrão chama o método chamado `self.visit_`*nomedaclasse* sendo *nomedaclasse* o nome da classe do nó, ou *generic_visit()* se aquele método não existir.

generic_visit(*node*)
Este visitante chama *visit()* em todos os filhos do nó.

Observe que nós filhos de nós que possuem um método de visitante personalizado não serão visitados, a menos que o visitante chame *generic_visit()* ou os visite por conta própria.

Não use o *NodeVisitor* se você quiser aplicar mudanças nos nós durante a travessia. Para isso existe um visitante especial (*NodeTransformer*) que permite modificações.

class `ast.`**NodeTransformer**
A subclasse *NodeVisitor* que percorre a árvore de sintaxe abstrata e permite a modificação de nós.

O *NodeTransformer* percorrerá a AST e usará o valor de retorno dos métodos do visitante para substituir ou remover o nó antigo. Se o valor de retorno do método visitante for `None`, o nó será removido de seu local, caso contrário, ele será substituído pelo valor de retorno. O valor de retorno pode ser o nó original, caso em que não há substituição.

Aqui está um exemplo de transformador que rescreve todas as ocorrências de procuras por nome (`foo`) para `data['foo']`:

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Index(value=Str(s=node.id)),
            ctx=node.ctx
        )
```

Tenha em mente que, se o nó em que você está operando tiver nós filhos, você deve transformar os nós filhos por conta própria ou chamar o método `generic_visit()` para o nó primeiro.

Para nós que faziam parte de uma coleção de instruções (que se aplica a todos os nós de instrução), o visitante também pode retornar uma lista de nós em vez de apenas um único nó.

Se *`NodeTransformer`* introduz novos nós (que não faziam parte da árvore original) sem fornecer informações de localização (como `lineno`), *`fix_missing_locations()`* deve ser chamado com o novo subárvore para recalcular as informações de localização:

```
tree = ast.parse('foo', mode='eval')
new_tree = fix_missing_locations(RewriteName().visit(tree))
```

Normalmente você usa o transformador assim:

```
node = YourTransformer().visit(node)
```

`ast.`**`dump`**(*node*, *annotate_fields=True*, *include_attributes=False*)
Retorne um despejo formatado da árvore em *node*. Isso é útil principalmente para fins de depuração. Se *annotate_fields* for verdadeiro (por padrão), a sequência retornada mostrará os nomes e os valores para os campos. Se *annotate_fields* for falso, a sequência de resultados será mais compacta ao omitir nomes de campos não ambíguos. Atributos como números de linha e deslocamentos de coluna não são despejados por padrão. Se isso for desejado, *include_attributes* pode ser definido como verdadeiro.

Ver também:

Green Tree Snakes, um recurso de documentação externo, possui bons detalhes sobre trabalhar com ASTs do Python.

33.3 `symtable` — Acesso a tabela de simbolos do compilador

Código Fonte: Lib/symtable.py

Symbol tables are generated by the compiler from AST just before bytecode is generated. The symbol table is responsible for calculating the scope of every identifier in the code. *`symtable`* provides an interface to examine these tables.

33.3.1 Generating Symbol Tables

symtable.**symtable**(*code*, *filename*, *compile_type*)
Return the toplevel *SymbolTable* for the Python source *code*. *filename* is the name of the file containing the code. *compile_type* is like the *mode* argument to *compile()*.

33.3.2 Examining Symbol Tables

class symtable.**SymbolTable**
A namespace table for a block. The constructor is not public.

get_type()
Return the type of the symbol table. Possible values are `'class'`, `'module'`, and `'function'`.

get_id()
Return the table's identifier.

get_name()
Return the table's name. This is the name of the class if the table is for a class, the name of the function if the table is for a function, or `'top'` if the table is global (*get_type()* returns `'module'`).

get_lineno()
Return the number of the first line in the block this table represents.

is_optimized()
Return `True` if the locals in this table can be optimized.

is_nested()
Return `True` if the block is a nested class or function.

has_children()
Return `True` if the block has nested namespaces within it. These can be obtained with *get_children()*.

has_exec()
Return `True` if the block uses `exec`.

get_identifiers()
Return a list of names of symbols in this table.

lookup(*name*)
Lookup *name* in the table and return a *Symbol* instance.

get_symbols()
Return a list of *Symbol* instances for names in the table.

get_children()
Return a list of the nested symbol tables.

class symtable.**Function**
A namespace for a function or method. This class inherits *SymbolTable*.

get_parameters()
Return a tuple containing names of parameters to this function.

get_locals()
Return a tuple containing names of locals in this function.

get_globals()
Return a tuple containing names of globals in this function.

get_frees()
Return a tuple containing names of free variables in this function.

class symtable.**Class**
A namespace of a class. This class inherits *SymbolTable*.

get_methods()
Return a tuple containing the names of methods declared in the class.

class symtable.**Symbol**
An entry in a *SymbolTable* corresponding to an identifier in the source. The constructor is not public.

get_name()
Return the symbol's name.

is_referenced()
Return `True` if the symbol is used in its block.

is_imported()
Return `True` if the symbol is created from an import statement.

is_parameter()
Return `True` if the symbol is a parameter.

is_global()
Return `True` if the symbol is global.

is_declared_global()
Return `True` if the symbol is declared global with a global statement.

is_local()
Return `True` if the symbol is local to its block.

is_free()
Return `True` if the symbol is referenced in its block, but not assigned to.

is_assigned()
Return `True` if the symbol is assigned to in its block.

is_namespace()
Return `True` if name binding introduces new namespace.

If the name is used as the target of a function or class statement, this will be true.

Por exemplo:

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

Note that a single name can be bound to multiple objects. If the result is `True`, the name may also be bound to other objects, like an int or list, that does not introduce a new namespace.

get_namespaces()
Return a list of namespaces bound to this name.

get_namespace()
Return the namespace bound to this name. If more than one namespace is bound, *ValueError* is raised.

33.4 symbol — Constantes usadas com árvores de análise do Python

Código Fonte: Lib/symbol.py

Este módulo fornece constantes que representam os valores numéricos dos nós internos da árvore de análise. Diferentemente da maioria das constantes do Python, elas usam nomes de letras minúsculas. Consulte o arquivo `Grammar/Grammar` na distribuição Python para as definições dos nomes no contexto da gramática do idioma. Os valores numéricos específicos para os quais os nomes mapeiam podem mudar entre as versões do Python.

Esse módulo também fornece um objeto de dados adicional.

`symbol.`**`sym_name`**
: Dicionário que mapeia os valores numéricos das constantes definidas neste módulo de volta para cadeias de nomes, permitindo que seja gerada uma representação mais legível de árvores de análise.

33.5 token — Constantes usadas com árvores de análises do Python

Código Fonte: Lib/token.py

Este módulo fornece constantes que representam os valores numéricos dos nós das folhas da árvore de análise (tokens terminais). Consulte o arquivo `Grammar/Grammar` na distribuição Python para obter as definições dos nomes no contexto da gramática da linguagem. Os valores numéricos específicos para os quais os nomes são mapeados podem mudar entre as versões do Python.

O módulo também fornece um mapeamento de códigos numéricos para nomes e algumas funções. As funções espelham definições nos arquivos de cabeçalho do Python C.

`token.`**`tok_name`**
: Dicionário que mapeia os valores numéricos das constantes definidas neste módulo de volta para cadeias de nomes, permitindo que seja gerada uma representação mais legível de árvores de análise.

`token.`**`ISTERMINAL`**`(`*x*`)`
: Retorna `True` para valores de tokens terminais.

`token.`**`ISNONTERMINAL`**`(`*x*`)`
: Retorna `True` para valores de tokens não terminais.

`token.`**`ISEOF`**`(`*x*`)`
: Retorna `True` se *x* for o marcador que indica o final da entrada.

Os constantes de tokens são:

`token.`**`ENDMARKER`**
`token.`**`NAME`**
`token.`**`NUMBER`**
`token.`**`STRING`**
`token.`**`NEWLINE`**
`token.`**`INDENT`**
`token.`**`DEDENT`**
`token.`**`LPAR`**
`token.`**`RPAR`**
`token.`**`LSQB`**

token.**RSQB**
token.**COLON**
token.**COMMA**
token.**SEMI**
token.**PLUS**
token.**MINUS**
token.**STAR**
token.**SLASH**
token.**VBAR**
token.**AMPER**
token.**LESS**
token.**GREATER**
token.**EQUAL**
token.**DOT**
token.**PERCENT**
token.**LBRACE**
token.**RBRACE**
token.**EQEQUAL**
token.**NOTEQUAL**
token.**LESSEQUAL**
token.**GREATEREQUAL**
token.**TILDE**
token.**CIRCUMFLEX**
token.**LEFTSHIFT**
token.**RIGHTSHIFT**
token.**DOUBLESTAR**
token.**PLUSEQUAL**
token.**MINEQUAL**
token.**STAREQUAL**
token.**SLASHEQUAL**
token.**PERCENTEQUAL**
token.**AMPEREQUAL**
token.**VBAREQUAL**
token.**CIRCUMFLEXEQUAL**
token.**LEFTSHIFTEQUAL**
token.**RIGHTSHIFTEQUAL**
token.**DOUBLESTAREQUAL**
token.**DOUBLESLASH**
token.**DOUBLESLASHEQUAL**
token.**AT**
token.**ATEQUAL**
token.**RARROW**
token.**ELLIPSIS**
token.**OP**
token.**ERRORTOKEN**
token.**N_TOKENS**
token.**NT_OFFSET**

Os seguintes valores de tipo de token não são usados pelo tokenizador do C, mas são necessários para o módulo *tokenize*.

token.**COMMENT**
Valor de token usado para indicar um comentário.

token.**NL**

Valor de token usado para indicar uma nova linha que não termina. O token *NEWLINE* indica o fim de uma linha lógica do código Python. Os tokens NL são gerados quando uma linha lógica de código é continuada em várias linhas físicas.

token.**ENCODING**
Valor de token que indica a codificação usada para decodificar os bytes de origem em texto. O primeiro token retornado por *tokenize.tokenize()* sempre será um token ENCODING.

Alterado na versão 3.5: Adicionados os tokens AWAIT e ASYNC.

Alterado na versão 3.7: Adicionados os tokens *COMMENT*, *NL* e *ENCODING*.

Alterado na versão 3.7: Removido os tokens AWAIT e ASYNC. "async" e "await" são agora tokenizados como tokens *NAME*.

33.6 keyword — Testando palavras-chave do Python

Código Fonte: Lib/keyword.py

Este módulo permite que um programa Python determine se uma string é uma palavra reservada.

keyword.**iskeyword**(*s*)
Retorna True se *s* for uma palavra reservad do Python.

keyword.**kwlist**
Sequência contendo todas as palavras reservadas definidas para o interpretador. Se alguma palavra reservada estiver definida para apenas estar ativa quando instruções *__future__* específicas tiverem efeito, estas serão incluídas também.

33.7 tokenize — Tokenizer for Python source

Código Fonte: Lib/tokenize.py

The *tokenize* module provides a lexical scanner for Python source code, implemented in Python. The scanner in this module returns comments as tokens as well, making it useful for implementing "pretty-printers", including colorizers for on-screen displays.

To simplify token stream handling, all operator and delimiter tokens and *Ellipsis* are returned using the generic *OP* token type. The exact type can be determined by checking the exact_type property on the *named tuple* returned from *tokenize.tokenize()*.

33.7.1 Tokenizando entradas

The primary entry point is a *generator*:

tokenize.**tokenize**(*readline*)
The *tokenize()* generator requires one argument, *readline*, which must be a callable object which provides the same interface as the *io.IOBase.readline()* method of file objects. Each call to the function should return one line of input as bytes.

The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple (srow, scol) of ints specifying the row and column where the token begins in the source; a 2-tuple (erow, ecol) of ints

specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed (the last tuple item) is the *logical* line; continuation lines are included. The 5 tuple is returned as a *named tuple* with the field names: `type string start end line`.

The returned *named tuple* has an additional property named `exact_type` that contains the exact operator type for `OP` tokens. For all other token types `exact_type` equals the named tuple `type` field.

Alterado na versão 3.1: Adiciona suporte para tuplas nomeadas.

Alterado na versão 3.3: Added support for `exact_type`.

`tokenize()` determines the source encoding of the file by looking for a UTF-8 BOM or encoding cookie, according to **PEP 263**.

All constants from the `token` module are also exported from `tokenize`.

Another function is provided to reverse the tokenization process. This is useful for creating tools that tokenize a script, modify the token stream, and write back the modified script.

`tokenize.`**`untokenize`**(*iterable*)
Converts tokens back into Python source code. The *iterable* must return sequences with at least two elements, the token type and the token string. Any additional sequence elements are ignored.

The reconstructed script is returned as a single string. The result is guaranteed to tokenize back to match the input so that the conversion is lossless and round-trips are assured. The guarantee applies only to the token type and token string as the spacing between tokens (column positions) may change.

It returns bytes, encoded using the `ENCODING` token, which is the first token sequence output by `tokenize()`.

`tokenize()` needs to detect the encoding of source files it tokenizes. The function it uses to do this is available:

`tokenize.`**`detect_encoding`**(*readline*)
The `detect_encoding()` function is used to detect the encoding that should be used to decode a Python source file. It requires one argument, readline, in the same way as the `tokenize()` generator.

It will call readline a maximum of twice, and return the encoding used (as a string) and a list of any lines (not decoded from bytes) it has read in.

It detects the encoding from the presence of a UTF-8 BOM or an encoding cookie as specified in **PEP 263**. If both a BOM and a cookie are present, but disagree, a `SyntaxError` will be raised. Note that if the BOM is found, `'utf-8-sig'` will be returned as an encoding.

If no encoding is specified, then the default of `'utf-8'` will be returned.

Use `open()` to open Python source files: it uses `detect_encoding()` to detect the file encoding.

`tokenize.`**`open`**(*filename*)
Open a file in read only mode using the encoding detected by `detect_encoding()`.

Novo na versão 3.2.

exception `tokenize.`**`TokenError`**
Raised when either a docstring or expression that may be split over several lines is not completed anywhere in the file, for example:

```
"""Beginning of
docstring
```

or:

```
[1,
 2,
 3
```

Note that unclosed single-quoted strings do not cause an error to be raised. They are tokenized as *ERRORTOKEN*, followed by the tokenization of their contents.

33.7.2 Uso da linha de comando

Novo na versão 3.3.

The *tokenize* module can be executed as a script from the command line. It is as simple as:

```
python -m tokenize [-e] [filename.py]
```

As seguintes opções são aceitas:

-h, --help
show this help message and exit

-e, --exact
display token names using the exact type

If `filename.py` is specified its contents are tokenized to stdout. Otherwise, tokenization is performed on stdin.

33.7.3 Exemplos

Example of a script rewriter that transforms float literals into Decimal objects:

```
from tokenize import tokenize, untokenize, NUMBER, STRING, NAME, OP
from io import BytesIO

def decistmt(s):
    """Substitute Decimals for floats in a string of statements.

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5*-.1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7'))"

    The format of the exponent is inherited from the platform C library.
    Known cases are "e-007" (Windows) and "e-07" (not Windows).  Since
    we're only showing 12 digits, and the 13th isn't close to 5, the
    rest of the output should be platform-independent.

    >>> exec(s)  #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    Output from calculations with Decimal should be identical across all
    platforms.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    """
    result = []
    g = tokenize(BytesIO(s.encode('utf-8')).readline)  # tokenize the string
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER tokens
            result.extend([
                (NAME, 'Decimal'),
```

(continua na próxima página)

(continuação da página anterior)

```
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
            result.append((toknum, tokval))
    return untokenize(result).decode('utf-8')
```

Example of tokenizing from the command line. The script:

```
def say_hello():
    print("Hello, World!")

say_hello()
```

will be tokenized to the following output where the first column is the range of the line/column coordinates where the token is found, the second column is the name of the token, and the final column is the value of the token (if any)

```
$ python -m tokenize hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          OP             '('
1,14-1,15:          OP             ')'
1,15-1,16:          OP             ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           OP             '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          OP             ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           OP             '('
4,10-4,11:          OP             ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

The exact token type names can be displayed using the `-e` option:

```
$ python -m tokenize -e hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          LPAR           '('
1,14-1,15:          RPAR           ')'
1,15-1,16:          COLON          ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           LPAR           '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          RPAR           ')'
2,26-2,27:          NEWLINE        '\n'
```

(continua na próxima página)

(continuação da página anterior)

```
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           LPAR           '('
4,10-4,11:          RPAR           ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

Example of tokenizing a file programmatically, reading unicode strings instead of bytes with `generate_tokens()`:

```
import tokenize

with tokenize.open('hello.py') as f:
    tokens = tokenize.generate_tokens(f.readline)
    for token in tokens:
        print(token)
```

Or reading bytes directly with *tokenize()*:

```
import tokenize

with open('hello.py', 'rb') as f:
    tokens = tokenize.tokenize(f.readline)
    for token in tokens:
        print(token)
```

33.8 tabnanny — Detecção de recuo ambíguo

Código Fonte: Lib/tabnanny.py

Por enquanto, este módulo deve ser chamado como um script. No entanto, é possível importá-lo para um IDE e usar a função *check()* descrita abaixo.

Nota: A API fornecida por este módulo provavelmente mudará em versões futuras; essas mudanças podem não ser compatíveis com versões anteriores.

---

tabnanny.**check**(*file_or_dir*)
: Se *file_or_dir* for um diretório e não um link simbólico, desce recursivamente a árvore de diretórios nomeada por *file_or_dir*, verificando todos os arquivos `.py` ao longo do caminho. Se *file_or_dir* for um arquivo-fonte comum do Python, ele será verificado quanto a problemas relacionados ao espaço em branco. As mensagens de diagnóstico são gravadas na saída padrão usando a função *print()*.

tabnanny.**verbose**
: Sinalizador indicando se as mensagens detalhadas devem ser impressas. Isso é incrementado pela opção `-v` se chamado como um script.

tabnanny.**filename_only**
: Sinalizador indicando se os nomes dos arquivos devem ser impressos apenas com problemas relacionados a espaços em branco. Isso é definido como true pela opção `-q` se chamado como um script.

exception tabnanny.**NannyNag**
: Levantada por *process_tokens()* se detectar um recuo ambíguo. Capturado e manipulado em *check()*.

tabnanny.**process_tokens**(*tokens*)
Esta função é usada por *check()* para processar os tokens gerados pelo módulo *tokenize*.

Ver também:

Módulo *tokenize* Scanner léxico para código fonte Python.

33.9 pyclbr — Suporte a navegador de módulos do Python

Código Fonte: Lib/pyclbr.py

O módulo *pyclbr* fornece informações limitadas sobre as funções, classes e métodos definidos em um módulo codificado em Python. As informações são suficientes para implementar um navegador de módulos. As informações são extraídas do código-fonte do Python em vez de importar o módulo, portanto, este módulo é seguro para uso com código não confiável. Essa restrição torna impossível o uso deste módulo com módulos não implementados no Python, incluindo todos os módulos de extensão padrão e opcionais.

pyclbr.**readmodule**(*module*, *path=None*)
Retorna um dicionário que mapeia os nomes de classe no nível do módulo aos descritores de classe. Se possível, descritores para classes base importadas estão incluídos. O parâmetro *module* é uma string com o nome do módulo a ser lido; pode ser o nome de um módulo dentro de um pacote. Se fornecido, *path* é uma sequência de caminhos de diretório anexada a `sys.path`, que é usada para localizar o código-fonte do módulo.

Esta função é a interface original e é mantida apenas para compatibilidade reversa. Ela retorna uma versão filtrada do seguinte.

pyclbr.**readmodule_ex**(*module*, *path=None*)
Retorna uma árvore baseada em dicionário que contém uma função ou descritores de classe para cada função e classe definida no módulo com uma declaração `def` ou `class`. O dicionário retornado mapeia os nomes das funções e das classes no nível do módulo para seus descritores. Objetos aninhados são inseridos no dicionário filho de seus pais. Como em readmodule, *module* nomeia o módulo a ser lido e *path* é anexado ao sys.path. Se o módulo que está sendo lido for um pacote, o dicionário retornado terá uma chave `'__path__'` cujo valor é uma lista que contém o caminho de pesquisa do pacote.

Novo na versão 3.7: Descritores para definições aninhadas. Eles são acessados através do novo atributo filho. Cada um tem um novo atributo pai.

Os descritores retornados por essas funções são instâncias das classes Function e Class. Não se espera que os usuários criem instâncias dessas classes.

33.9.1 Objetos de Função

Instâncias da classe :class:Function‘ descrevem funções definidas por instruções def. Eles têm os seguintes atributos:

Function.**file**
Nome do arquivo no qual a função está definida.

Function.**module**
O nome do módulo que define a função descrita.

Function.**name**
O nome da função.

Function.**lineno**
O número da linha no arquivo em que a definição é iniciada.

`Function.`**`parent`**
Para funções de nível superior, Nenhuma. Para funções aninhadas, o pai.

Novo na versão 3.7.

`Function.`**`children`**
Um dicionário que mapeia nomes para descritores para funções e classes aninhadas.

Novo na versão 3.7.

33.9.2 Objetos de Classe

Instâncias da classe :class:Class‘ descrevem classes definidas por instruções class. Elas têm os mesmos atributos que Functions e mais dois.

`Class.`**`file`**
Nome do arquivo no qual a classe está definida.

`Class.`**`module`**
O nome do módulo que define a classe descrita.

`Class.`**`name`**
O nome da turma.

`Class.`**`lineno`**
O número da linha no arquivo em que a definição é iniciada.

`Class.`**`parent`**
Para classes de nível superior, None. Para classes aninhadas, o pai.

Novo na versão 3.7.

`Class.`**`children`**
Um dicionário que mapeia nomes para descritores para funções e classes aninhadas.

Novo na versão 3.7.

`Class.`**`super`**
Uma lista de objetos `Class` que descreve as classes base imediatas da classe que está sendo descrita. Classes nomeadas como superclasses, mas que não podem ser descobertas por *`readmodule_ex()`* são listadas como uma string com o nome da classe em vez de como objetos de `Class`.

`Class.`**`methods`**
Um dicionário que mapeia nomes de métodos para números de linha. Isso pode ser derivado do dicionário filho mais novo, mas permanece para compatibilidade retroativa.

33.10 `py_compile` — Compilar arquivos fonte do Python

Código-Fonte: Lib/py_compile.py

O módulo *`py_compile`* fornece uma função para gerar um arquivo de bytecode a partir de um arquivo fonte, e outra função usada quando o arquivo fonte do módulo é chamado como um script.

Embora nem sempre seja necessária, essa função pode ser útil ao instalar módulos para uso compartilhado, especialmente se alguns usuários não tiverem permissão para gravar os arquivos de cache de bytecodes no diretório que contém o código-fonte.

exception py_compile.**PyCompileError**
Exceção levantada quando ocorre um erro ao tentar compilar o arquivo.

py_compile.**compile**(*file*, *cfile=None*, *dfile=None*, *doraise=False*, *optimize=-1*, *invalidation_mode=PycInvalidationMode.TIMESTAMP*)
Compila um arquivo fonte para bytecode e grava o arquivo de cache de bytecode. O código-fonte é carregado a partir do arquivo chamado *file*. O bytecode é gravado em *cfile*, cujo padrão é o caminho **PEP 3147/PEP 488**, terminando em `.pyc`. Por exemplo, se *file* for `/foo/bar/baz.py`, o *cfile* será padronizado como `/foo/bar/__pycache__/baz.cpython-32.pyc` para o Python 3.2. Se *dfile* for especificado, ele será usado como o nome do arquivo fonte nas mensagens de erro quando em vez de *file*. Se *doraise* for verdadeiro, a *PyCompileError* será levantada quando um erro for encontrado durante a compilação de *file*. Se *doraise* for false (o padrão), uma string de erros será gravada em `sys.stderr`, mas nenhuma exceção será gerada. Essa função retorna o caminho para o arquivo compilado em bytes, ou seja, qualquer valor *cfile* foi usado.

Se o caminho que *cfile* se tornar (especificado ou computado explicitamente) for um link simbólico ou um arquivo não regular, *FileExistsError* será levantada. Isso serve como um aviso de que a importação transformará esses caminhos em arquivos regulares se for permitido gravar arquivos compilados em bytes nesses caminhos. Esse é um efeito colateral da importação usando a renomeação de arquivo para colocar o arquivo final compilado em bytecode para evitar problemas de gravação simultânea de arquivos.

optimize controla o nível de otimização e é passado para a função interna *compile()*. O padrão de `-1` seleciona o nível de otimização do interpretador atual.

invalidation_mode deve ser um membro da enum *PycInvalidationMode* e controla como o cache do bytecode gerado é invalidado em tempo de execução. O padrão é *PycInvalidationMode.CHECKED_HASH* se a variável de ambiente `SOURCE_DATE_EPOCH` estiver configurada, caso contrário, o padrão é *PycInvalidationMode.TIMESTAMP*.

Alterado na versão 3.2: Alterado o valor padrão de *cfile* para ficar em conformidade com a **PEP 3147**. O padrão anterior era *file* + `'c'` (`'o'` se a otimização estivesse ativada). Também foi adicionado o parâmetro *optimize*.

Alterado na versão 3.4: Alterado o código para usar *importlib* para a gravação do arquivo de cache do bytecode. Isso significa que a semântica de criação/gravação de arquivo agora corresponde ao que *importlib* faz, por exemplo, permissões, semântica de gravação e movimentação, etc. Também foi adicionada a ressalva de que *FileExistsError* é levantada se *cfile* for um link simbólico ou um arquivo não regular.

Alterado na versão 3.7: O parâmetro *invalidation_mode* foi adicionado conforme especificado em **PEP 552**. Se a variável de ambiente `SOURCE_DATE_EPOCH` estiver configurada, *invalidation_mode* será forçado a *PycInvalidationMode.CHECKED_HASH*.

Alterado na versão 3.7.2: A variável de ambiente `SOURCE_DATE_EPOCH` não substitui mais o valor do argumento *invalidation_mode* e, em vez disso, determina seu valor padrão.

class py_compile.**PycInvalidationMode**
Uma enumeração de métodos possíveis que o interpretador pode usar para determinar se um arquivo de bytecode está atualizado com um arquivo fonte. O arquivo `.pyc` indica o modo de invalidação desejado em seu cabeçalho. Veja pyc-invalidation para obter mais informações sobre como o Python invalida arquivos `.pyc` em tempo de execução.

Novo na versão 3.7.

TIMESTAMP
O arquivo `.pyc` inclui o carimbo de data e hora e o tamanho do arquivo fonte, que o Python comparará com os metadados do arquivo fonte no tempo de execução para determinar se o arquivo `.pyc` precisa ser gerado novamente.

CHECKED_HASH
O arquivo `.pyc` inclui um hash do conteúdo do arquivo fonte, com o qual o Python comparará o fonte em tempo de execução para determinar se o arquivo `.pyc` precisa ser gerado novamente.

UNCHECKED_HASH
Como *CHECKED_HASH*, o arquivo `.pyc` inclui um hash do conteúdo do arquivo fonte. No entanto, em tempo de execução, o Python presumirá que o arquivo `.pyc` está atualizado e não validará o `.pyc` contra o arquivo fonte.

Essa opção é útil quando os `.pycs` são atualizados por algum sistema externo ao Python, como um sistema de compilação.

py_compile.**main**(*args=None*)
Compila vários arquivos fonte. Os arquivos nomeados em *args* (ou na linha de comando, se *args* for `None`) são compilados e o bytecode resultante é armazenado em cache da maneira normal. Esta função não pesquisa uma estrutura de diretórios para localizar arquivos fonte; ela apenas compila arquivos nomeados explicitamente. Se `'-'` é o único parâmetro em args, a lista de arquivos é obtida da entrada padrão.

Alterado na versão 3.2: Adicionado suporte a `'-'`.

Quando este módulo é executado como um script, *main()* é usada para compilar todos os arquivos nomeados na linha de comando. O status de saída é diferente de zero se um dos arquivos não pôde ser compilado.

Ver também:

Módulo *compileall* Utilitários para compilar todos os arquivos fontes Python em uma árvore de diretórios.

33.11 compileall — Compilar bibliotecas do Python para bytecode

Código Fonte: Lib/compileall.py

Este módulo fornece algumas funções utilitárias para dar suporte à instalação de bibliotecas Python. Essas funções compilam arquivos fonte Python em uma árvore de diretórios. Este módulo pode ser usado para criar os arquivos de bytecodes em cache no momento da instalação da biblioteca, o que os torna disponíveis para uso mesmo por usuários que não têm permissão de gravação nos diretórios da biblioteca.

33.11.1 Uso na linha de comando

Este módulo pode funcionar como um script (usando **python -m compileall**) para compilar fontes do Python.

directory ...
file ...
Argumentos posicionais são arquivos a serem compilados ou diretórios que contêm arquivos de origem, percorridos recursivamente. Se nenhum argumento for fornecido, comporta-se como se a linha de comando fosse `-l <diretórios do sys.path>`.

-l
Não atua recursivamente em subdiretórios, apenas compila arquivos de código-fonte diretamente contidos nos diretórios nomeados ou implícitos.

-f
Força a recompilação, mesmo que os carimbos de data e hora estejam atualizados.

-q
Não imprime a lista de arquivos compilados. Se passado uma vez, as mensagens de erro ainda serão impressas. Se passado duas vezes (`-qq`), toda a saída é suprimida.

-d destdir
: Diretório anexado ao caminho de cada arquivo que está sendo compilado. Isso aparecerá nos tracebacks em tempo de compilação e também será compilado no arquivo de bytecode, onde será usado em tracebacks e outras mensagens nos casos em que o arquivo de origem não exista no momento em que o arquivo de bytecode for executado.

-x regex
: A expressão regular `regex` é usada para pesquisar o caminho completo para cada arquivo considerado para compilação e, se a `regex` produzir uma correspondência, o arquivo será ignorado.

-i list
: Lê o arquivo `list` e adicione cada linha que ele contém à lista de arquivos e diretórios a serem compilados. Se `list` for `-`, lê as linhas do `stdin`.

-b
: Escreve os arquivos de bytecode em seus locais e nomes legados, que podem sobrescrever arquivos de bytecode criados por outra versão do Python. O padrão é gravar arquivos em seus locais e nomes do **PEP 3147**, o que permite que arquivos de bytecode de várias versões do Python coexistam.

-r
: Controla o nível máximo de recursão para subdiretórios. Se isso for dado, a opção `-l` não será levada em consideração. **python -m compileall <diretório> -r 0** é equivalente a **python -m compileall <diretório> -l**.

-j N
: Use *N* workers para compilar os arquivos dentro do diretório especificado. Se `0` for usado, o resultado de *os.cpu_count()* será usado.

--invalidation-mode [timestamp|checked-hash|unchecked-hash]
: Controla como os arquivos de bytecode gerados são invalidados no tempo de execução. O valor `timestamp` significa que os arquivos `.pyc` com o carimbo de data/hora do fonte e o tamanho incorporado serão gerados. Os valores `selected-hash` e `unchecked-hash` fazem com que os pycs baseados em hash sejam gerados. Arquivos pycs baseados em hash incorporam um hash do conteúdo do arquivo fonte em vez de um carimbo de data/hora. Veja pyc-invalidation para obter mais informações sobre como o Python valida os arquivos de cache do bytecode em tempo de execução. O padrão é `timestamp` se a variável de ambiente `SOURCE_DATE_EPOCH` não estiver configurada e `selected-hash` se a variável de ambiente `SOURCE_DATE_EPOCH` estiver configurada.

Alterado na versão 3.2: Adicionadas as opções `-i`, `-b` e `-h`.

Alterado na versão 3.5: Adicionadas as opções `-j`, `-r` e `-qq`. A opção `-q` foi alterada para um valor multinível. `-b` sempre produzirá um arquivo de bytecodes que termina em `.pyc`, nunca em `.pyo`.

Alterado na versão 3.7: Adicionada a opção `--invalidation-mode`.

Não há opção na linha de comando para controlar o nível de otimização usado pela função *compile()* porque o próprio interpretador Python já fornece a opção: **python -O -m compileall**.

33.11.2 Funções públicas

compileall.**compile_dir**(*dir*, *maxlevels=10*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *workers=1*, *invalidation_mode=py_compile.PycInvalidationMode.TIMESTAMP*)
: Desce recursivamente a árvore de diretórios nomeada por *dir*, compilando todos os arquivos `.py` ao longo do caminho. Retorna um valor verdadeiro se todos os arquivos forem compilados com êxito e um valor falso caso contrário.

  O parâmetro *maxlevels* é usado para limitar a profundidade da recursão; o padrão é `10`.

  Se *ddir* for fornecido, ele será anexado ao caminho de cada arquivo que está sendo compilado para uso em tracebacks em tempo de compilação e também será compilado no arquivo de bytecode, onde será usado em tracebacks

e outras mensagens nos casos em que o arquivo de origem não existe no momento em que o arquivo de bytecode é executado.

Se *force* for verdadeiro, os módulos serão recompilados, mesmo que os carimbos de data e hora estejam atualizados.

Se *rx* for fornecido, seu método de pesquisa será chamado no caminho completo para cada arquivo considerado para compilação e, se retornar um valor verdadeiro, o arquivo será ignorado.

Se *quiet* for `False` ou `0` (o padrão), os nomes dos arquivos e outras informações serão impressos com o padrão. Definido como `1`, apenas os erros são impressos. Definido como `2`, toda a saída é suprimida.

Se *legacy* for verdadeiro, os arquivos de bytecodes serão gravados em seus locais e nomes herdados, o que poderá sobrescrever arquivos de bytecodes criados por outra versão do Python. O padrão é gravar arquivos em seus locais e nomes do **PEP 3147**, o que permite que arquivos de bytecodes de várias versões do Python coexistam.

optimize especifica o nível de otimização para o compilador. Ele é passado para a função embutida `compile()`.

The argument *workers* specifies how many workers are used to compile files in parallel. The default is to not use multiple workers. If the platform can't use multiple workers and *workers* argument is given, then sequential compilation will be used as a fallback. If *workers* is lower than `0`, a `ValueError` will be raised.

invalidation_mode deve ser um membro de enum `py_compile.PycInvalidationMode` e controla como os pycs gerados são invalidados em tempo de execução.

Alterado na versão 3.2: Adicionado os parâmetros *legacy* e *optimize*.

Alterado na versão 3.5: Adicionado o parâmetro *workers*.

Alterado na versão 3.5: O parâmetro *quiet* foi alterado para um valor multinível.

Alterado na versão 3.5: O parâmetro *legacy* grava apenas arquivos `.pyc`, não os arquivos `.pyo`, independentemente do valor de *optimize*.

Alterado na versão 3.6: Aceita um *path-like object*.

Alterado na versão 3.7: O parâmetro *invalidation_mode* foi adicionado.

`compileall.`**`compile_file`**(*fullname*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=py_compile.PycInvalidationMode.TIMESTAMP*)

Compila o arquivo com o caminho *fullname*. Retorna um valor verdadeiro se o arquivo compilado com êxito e um valor falso caso contrário.

Se *ddir* for fornecido, ele será anexado ao caminho do arquivo que está sendo compilado para uso em rastreamentos em tempo de compilação e também será compilado no arquivo de bytecode, onde será usado em tracebacks e outras mensagens nos casos em que o arquivo fonte não existe no momento em que o arquivo de bytecode é executado.

Se *rx* for fornecido, seu método de pesquisa passará o nome do caminho completo para o arquivo que está sendo compilado e, se retornar um valor verdadeiro, o arquivo não será compilado e `True` será retornado.

Se *quiet* for `False` ou `0` (o padrão), os nomes dos arquivos e outras informações serão impressos com o padrão. Definido como `1`, apenas os erros são impressos. Definido como `2`, toda a saída é suprimida.

Se *legacy* for verdadeiro, os arquivos de bytecodes serão gravados em seus locais e nomes herdados, o que poderá sobrescrever arquivos de bytecodes criados por outra versão do Python. O padrão é gravar arquivos em seus locais e nomes do **PEP 3147**, o que permite que arquivos de bytecodes de várias versões do Python coexistam.

optimize especifica o nível de otimização para o compilador. Ele é passado para a função embutida `compile()`.

invalidation_mode deve ser um membro de enum `py_compile.PycInvalidationMode` e controla como os pycs gerados são invalidados em tempo de execução.

Novo na versão 3.2.

Alterado na versão 3.5: O parâmetro *quiet* foi alterado para um valor multinível.

Alterado na versão 3.5: O parâmetro *legacy* grava apenas arquivos `.pyc`, não os arquivos `.pyo`, independentemente do valor de *optimize*.

Alterado na versão 3.7: O parâmetro *invalidation_mode* foi adicionado.

`compileall.`**`compile_path`**(*skip_curdir=True*, *maxlevels=0*, *force=False*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=py_compile.PycInvalidationMode.TIMESTAMP*)

Compila Byte para bytecodes todos os arquivos `.py` encontrados ao longo de `sys.path`. Retorna um valor verdadeiro se todos os arquivos forem compilados com êxito e um valor falso caso contrário.

Se *skip_curdir* for verdadeiro (o padrão), o diretório atual não será incluído na pesquisa. Todos os outros parâmetros são passados para a função *compile_dir()*. Note que, ao contrário das outras funções de compilação, `maxlevels` é padronizado como `0`.

Alterado na versão 3.2: Adicionado os parâmetros *legacy* e *optimize*.

Alterado na versão 3.5: O parâmetro *quiet* foi alterado para um valor multinível.

Alterado na versão 3.5: O parâmetro *legacy* grava apenas arquivos `.pyc`, não os arquivos `.pyo`, independentemente do valor de *optimize*.

Alterado na versão 3.7: O parâmetro *invalidation_mode* foi adicionado.

Para forçar uma recompilação de todos os arquivos `.py` no subdiretório `Lib/` e todos os seus subdiretórios:

```
import compileall

compileall.compile_dir('Lib/', force=True)

# Perform same compilation, excluding files in .svn directories.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\][.]svn'), force=True)

# pathlib.Path objects can also be used.
import pathlib
compileall.compile_dir(pathlib.Path('Lib/'), force=True)
```

Ver também:

Módulo *py_compile* Compila para bytecode um único arquivo fonte.

33.12 `dis` — Disassembler do bytecode do Python

Código Fonte: Lib/dis.py

O módulo *dis* suporta a análise dos termos *bytecode* CPython, desmontando-o. O bytecode do CPython que o módulo leva como entrada é definido no arquivo arquivoi `Incluir/opcode.h` e usado pelo compilador e pelo intérprete.

CPython implementation detail: O Bytecode é um detalhe de implementação do intérprete do CPython. Não há garantias de que o bytecode não será adicionado, removido ou alterado entre as versões do Python. O uso deste módulo não deve ser considerado que funcionará em todas as VMs do Python ou mesmo em verões futuras.

Alterado na versão 3.6: Use 2 bytes para cada instrução. Anteriormente, o número de bytes variava de acordo com as instruções.

Exemplo: Dada a função `myfunc()`:

```
def myfunc(alist):
    return len(alist)
```

o seguinte comando pode ser usado para exibir a desmontagem da função `myfunc()`:

```
>>> dis.dis(myfunc)
  2           0 LOAD_GLOBAL              0 (len)
              2 LOAD_FAST                0 (alist)
              4 CALL_FUNCTION            1
              6 RETURN_VALUE
```

(O "2" é um número da linha).

33.12.1 Analise do Bytecode

Novo na versão 3.4.

A API de análise de bytecode permite que partes do código Python sejam Wrapped em um objeto da *Bytecode* que facilite o acesso aos detalhes do código compilado.

class dis.**Bytecode**(*x*, *, *first_line=None*, *current_offset=None*)

Analyse the bytecode corresponding to a function, generator, asynchronous generator, coroutine, method, string of source code, or a code object (as returned by *compile()*).

Este é um Wrapper de conveniência em torno de muitas das funções listadas abaixo, mais notavelmente a funçõa *get_instructions()*, como iterando sobre uma instância *Bytecode* produz as operações bytecode como nas instância *Instruction*.

If *first_line* is not `None`, it indicates the line number that should be reported for the first source line in the disassembled code. Otherwise, the source line information (if any) is taken directly from the disassembled code object.

If *current_offset* is not `None`, it refers to an instruction offset in the disassembled code. Setting this means *dis()* will display a "current instruction" marker against the specified opcode.

classmethod from_traceback(*tb*)

Construct a *Bytecode* instance from the given traceback, setting *current_offset* to the instruction responsible for the exception.

codeobj

The compiled code object.

first_line

The first source line of the code object (if available)

dis()

Return a formatted view of the bytecode operations (the same as printed by *dis.dis()*, but returned as a multi-line string).

info()

Return a formatted multi-line string with detailed information about the code object, like *code_info()*.

Alterado na versão 3.7: This can now handle coroutine and asynchronous generator objects.

Exemplo:

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
```

(continua na próxima página)

(continuação da página anterior)

```
...
LOAD_GLOBAL
LOAD_FAST
CALL_FUNCTION
RETURN_VALUE
```

33.12.2 Analysis functions

The *dis* module also defines the following analysis functions that convert the input directly to the desired output. They can be useful if only a single operation is being performed, so the intermediate analysis object isn't useful:

dis.**code_info**(*x*)
Return a formatted multi-line string with detailed code object information for the supplied function, generator, asynchronous generator, coroutine, method, source code string or code object.

Note that the exact contents of code info strings are highly implementation dependent and they may change arbitrarily across Python VMs or Python releases.

Novo na versão 3.2.

Alterado na versão 3.7: This can now handle coroutine and asynchronous generator objects.

dis.**show_code**(*x*, *, *file=None*)
Print detailed code object information for the supplied function, method, source code string or code object to *file* (or `sys.stdout` if *file* is not specified).

This is a convenient shorthand for `print(code_info(x), file=file)`, intended for interactive exploration at the interpreter prompt.

Novo na versão 3.2.

Alterado na versão 3.4: Added *file* parameter.

dis.**dis**(*x=None*, *, *file=None*, *depth=None*)
Disassemble the *x* object. *x* can denote either a module, a class, a method, a function, a generator, an asynchronous generator, a coroutine, a code object, a string of source code or a byte sequence of raw bytecode. For a module, it disassembles all functions. For a class, it disassembles all methods (including class and static methods). For a code object or sequence of raw bytecode, it prints one line per bytecode instruction. It also recursively disassembles nested code objects (the code of comprehensions, generator expressions and nested functions, and the code used for building nested classes). Strings are first compiled to code objects with the *compile()* built-in function before being disassembled. If no object is provided, this function disassembles the last traceback.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

The maximal depth of recursion is limited by *depth* unless it is `None`. `depth=0` means no recursion.

Alterado na versão 3.4: Added *file* parameter.

Alterado na versão 3.7: Implemented recursive disassembling and added *depth* parameter.

Alterado na versão 3.7: This can now handle coroutine and asynchronous generator objects.

dis.**distb**(*tb=None*, *, *file=None*)
Disassemble the top-of-stack function of a traceback, using the last traceback if none was passed. The instruction causing the exception is indicated.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

Alterado na versão 3.4: Added *file* parameter.

dis.**disassemble**(*code*, *lasti=-1*, *, *file=None*)

dis.**disco**(*code*, *lasti=-1*, *, *file=None*)
Disassemble a code object, indicating the last instruction if *lasti* was provided. The output is divided in the following columns:

1. the line number, for the first instruction of each line
2. the current instruction, indicated as `-->`,
3. a labelled instruction, indicated with `>>`,
4. the address of the instruction,
5. the operation code name,
6. operation parameters, and
7. interpretation of the parameters in parentheses.

The parameter interpretation recognizes local and global variable names, constant values, branch targets, and compare operators.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

Alterado na versão 3.4: Added *file* parameter.

dis.**get_instructions**(*x*, *, *first_line=None*)
Return an iterator over the instructions in the supplied function, method, source code string or code object.

The iterator generates a series of *Instruction* named tuples giving the details of each operation in the supplied code.

If *first_line* is not `None`, it indicates the line number that should be reported for the first source line in the disassembled code. Otherwise, the source line information (if any) is taken directly from the disassembled code object.

Novo na versão 3.4.

dis.**findlinestarts**(*code*)
This generator function uses the `co_firstlineno` and `co_lnotab` attributes of the code object *code* to find the offsets which are starts of lines in the source code. They are generated as `(offset, lineno)` pairs. See Objects/lnotab_notes.txt for the `co_lnotab` format and how to decode it.

Alterado na versão 3.6: Line numbers can be decreasing. Before, they were always increasing.

dis.**findlabels**(*code*)
Detect all offsets in the raw compiled bytecode string *code* which are jump targets, and return a list of these offsets.

dis.**stack_effect**(*opcode*[, *oparg*])
Compute the stack effect of *opcode* with argument *oparg*.

Novo na versão 3.4.

33.12.3 Python Bytecode Instructions

The *get_instructions()* function and *Bytecode* class provide details of bytecode instructions as *Instruction* instances:

class dis.**Instruction**
Details for a bytecode operation

opcode
numeric code for operation, corresponding to the opcode values listed below and the bytecode values in the *Opcode collections*.

opname
human readable name for operation

arg
numeric argument to operation (if any), otherwise `None`

argval
resolved arg value (if known), otherwise same as arg

argrepr
human readable description of operation argument

offset
start index of operation within bytecode sequence

starts_line
line started by this opcode (if any), otherwise `None`

is_jump_target
`True` if other code jumps to here, otherwise `False`

Novo na versão 3.4.

The Python compiler currently generates the following bytecode instructions.

General instructions

NOP
Do nothing code. Used as a placeholder by the bytecode optimizer.

POP_TOP
Removes the top-of-stack (TOS) item.

ROT_TWO
Swaps the two top-most stack items.

ROT_THREE
Lifts second and third stack item one position up, moves top down to position three.

DUP_TOP
Duplicates the reference on top of the stack.

Novo na versão 3.2.

DUP_TOP_TWO
Duplicates the two references on top of the stack, leaving them in the same order.

Novo na versão 3.2.

Unary operations

Unary operations take the top of the stack, apply the operation, and push the result back on the stack.

UNARY_POSITIVE
Implements `TOS = +TOS`.

UNARY_NEGATIVE
Implements `TOS = -TOS`.

UNARY_NOT
Implements `TOS = not TOS`.

UNARY_INVERT
Implementação `TOS = ~TOS`.

GET_ITER
Implementa `TOS = iter(TOS)`.

GET_YIELD_FROM_ITER
If `TOS` is a *generator iterator* or *coroutine* object it is left as is. Otherwise, implements `TOS = iter(TOS)`.

Novo na versão 3.5.

Operações Binárias

Binary operations remove the top of the stack (TOS) and the second top-most stack item (TOS1) from the stack. They perform the operation, and put the result back on the stack.

BINARY_POWER
Implementa `TOS = TOS1 ** TOS`.

BINARY_MULTIPLY
Implements `TOS = TOS1 * TOS`.

BINARY_MATRIX_MULTIPLY
Implementado `TOS = TOS1 @ TOS`.

Novo na versão 3.5.

BINARY_FLOOR_DIVIDE
Implementa `TOS = TOS1 // TOS`.

BINARY_TRUE_DIVIDE
Implementa `TOS = TOS1 / TOS`.

BINARY_MODULO
Implementa `TOS = TOS1 % TOS`.

BINARY_ADD
Implements `TOS = TOS1 + TOS`.

BINARY_SUBTRACT
Implements `TOS = TOS1 - TOS`.

BINARY_SUBSCR
Implements `TOS = TOS1[TOS]`.

BINARY_LSHIFT
Implements `TOS = TOS1 << TOS`.

BINARY_RSHIFT
Implements `TOS = TOS1 >> TOS`.

BINARY_AND
Implements `TOS = TOS1 & TOS`.

BINARY_XOR
Implements `TOS = TOS1 ^ TOS`.

BINARY_OR
Implements `TOS = TOS1 | TOS`.

In-place operations

In-place operations are like binary operations, in that they remove TOS and TOS1, and push the result back on the stack, but the operation is done in-place when TOS1 supports it, and the resulting TOS may be (but does not have to be) the original TOS1.

INPLACE_POWER
Implements in-place `TOS = TOS1 ** TOS`.

INPLACE_MULTIPLY
Implements in-place `TOS = TOS1 * TOS`.

INPLACE_MATRIX_MULTIPLY
Implements in-place `TOS = TOS1 @ TOS`.

Novo na versão 3.5.

INPLACE_FLOOR_DIVIDE
Implements in-place `TOS = TOS1 // TOS`.

INPLACE_TRUE_DIVIDE
Implements in-place `TOS = TOS1 / TOS`.

INPLACE_MODULO
Implements in-place `TOS = TOS1 % TOS`.

INPLACE_ADD
Implements in-place `TOS = TOS1 + TOS`.

INPLACE_SUBTRACT
Implements in-place `TOS = TOS1 - TOS`.

INPLACE_LSHIFT
Implements in-place `TOS = TOS1 << TOS`.

INPLACE_RSHIFT
Implements in-place `TOS = TOS1 >> TOS`.

INPLACE_AND
Implements in-place `TOS = TOS1 & TOS`.

INPLACE_XOR
Implements in-place `TOS = TOS1 ^ TOS`.

INPLACE_OR
Implements in-place `TOS = TOS1 | TOS`.

STORE_SUBSCR
Implements `TOS1[TOS] = TOS2`.

DELETE_SUBSCR
Implements `del TOS1[TOS]`.

Coroutine opcodes

GET_AWAITABLE
Implements `TOS = get_awaitable(TOS)`, where `get_awaitable(o)` returns `o` if `o` is a coroutine object or a generator object with the CO_ITERABLE_COROUTINE flag, or resolves `o.__await__`.

Novo na versão 3.5.

GET_AITER
Implements `TOS = TOS.__aiter__()`.

Novo na versão 3.5.

Alterado na versão 3.7: Returning awaitable objects from `__aiter__` is no longer supported.

GET_ANEXT
Implements `PUSH(get_awaitable(TOS.__anext__()))`. See `GET_AWAITABLE` for details about `get_awaitable`

Novo na versão 3.5.

BEFORE_ASYNC_WITH
Resolves `__aenter__` and `__aexit__` from the object on top of the stack. Pushes `__aexit__` and result of `__aenter__()` to the stack.

Novo na versão 3.5.

SETUP_ASYNC_WITH
Creates a new frame object.

Novo na versão 3.5.

Miscellaneous opcodes

PRINT_EXPR
Implements the expression statement for the interactive mode. TOS is removed from the stack and printed. In non-interactive mode, an expression statement is terminated with *POP_TOP*.

BREAK_LOOP
Terminates a loop due to a `break` statement.

CONTINUE_LOOP (*target*)
Continues a loop due to a `continue` statement. *target* is the address to jump to (which should be a *FOR_ITER* instruction).

SET_ADD (*i*)
Calls `set.add(TOS1[-i], TOS)`. Used to implement set comprehensions.

LIST_APPEND (*i*)
Calls `list.append(TOS[-i], TOS)`. Used to implement list comprehensions.

MAP_ADD (*i*)
Calls `dict.setitem(TOS1[-i], TOS, TOS1)`. Used to implement dict comprehensions.

Novo na versão 3.1.

For all of the *SET_ADD*, *LIST_APPEND* and *MAP_ADD* instructions, while the added value or key/value pair is popped off, the container object remains on the stack so that it is available for further iterations of the loop.

RETURN_VALUE
Returns with TOS to the caller of the function.

YIELD_VALUE
Pops TOS and yields it from a *generator*.

YIELD_FROM
Pops TOS and delegates to it as a subiterator from a *generator*.

Novo na versão 3.3.

SETUP_ANNOTATIONS
Checks whether `__annotations__` is defined in `locals()`, if not it is set up to an empty `dict`. This opcode is only emitted if a class or module body contains *variable annotations* statically.

Novo na versão 3.6.

IMPORT_STAR
Loads all symbols not starting with `'_'` directly from the module TOS to the local namespace. The module is popped after loading all names. This opcode implements `from module import *`.

POP_BLOCK
Removes one block from the block stack. Per frame, there is a stack of blocks, denoting nested loops, try statements, and such.

POP_EXCEPT
Removes one block from the block stack. The popped block must be an exception handler block, as implicitly created when entering an except handler. In addition to popping extraneous values from the frame stack, the last three popped values are used to restore the exception state.

END_FINALLY
Terminates a `finally` clause. The interpreter recalls whether the exception has to be re-raised, or whether the function returns, and continues with the outer-next block.

LOAD_BUILD_CLASS
Pushes `builtins.__build_class__()` onto the stack. It is later called by *CALL_FUNCTION* to construct a class.

SETUP_WITH (*delta*)
This opcode performs several operations before a with block starts. First, it loads `__exit__()` from the context manager and pushes it onto the stack for later use by `WITH_CLEANUP`. Then, `__enter__()` is called, and a finally block pointing to *delta* is pushed. Finally, the result of calling the enter method is pushed onto the stack. The next opcode will either ignore it (*POP_TOP*), or store it in (a) variable(s) (*STORE_FAST*, *STORE_NAME*, or *UNPACK_SEQUENCE*).

Novo na versão 3.2.

WITH_CLEANUP_START
Cleans up the stack when a `with` statement block exits. TOS is the context manager's `__exit__()` bound method. Below TOS are 1–3 values indicating how/why the finally clause was entered:

- SECOND = `None`
- (SECOND, THIRD) = (`WHY_{RETURN,CONTINUE}`), retval
- SECOND = `WHY_*`; no retval below it
- (SECOND, THIRD, FOURTH) = exc_info()

In the last case, `TOS(SECOND, THIRD, FOURTH)` is called, otherwise `TOS(None, None, None)`. Pushes SECOND and result of the call to the stack.

WITH_CLEANUP_FINISH
Pops exception type and result of 'exit' function call from the stack.

If the stack represents an exception, *and* the function call returns a 'true' value, this information is "zapped" and replaced with a single `WHY_SILENCED` to prevent *END_FINALLY* from re-raising the exception. (But non-local gotos will still be resumed.)

All of the following opcodes use their arguments.

STORE_NAME (*namei*)
Implements `name = TOS`. *namei* is the index of *name* in the attribute `co_names` of the code object. The compiler tries to use *STORE_FAST* or *STORE_GLOBAL* if possible.

DELETE_NAME (*namei*)
Implements `del name`, where *namei* is the index into `co_names` attribute of the code object.

UNPACK_SEQUENCE (*count*)
Unpacks TOS into *count* individual values, which are put onto the stack right-to-left.

UNPACK_EX (*counts*)
Implements assignment with a starred target: Unpacks an iterable in TOS into individual values, where the total number of values can be smaller than the number of items in the iterable: one of the new values will be a list of all leftover items.

The low byte of *counts* is the number of values before the list value, the high byte of *counts* the number of values after it. The resulting values are put onto the stack right-to-left.

STORE_ATTR(*namei*)
Implements `TOS.name = TOS1`, where *namei* is the index of name in `co_names`.

DELETE_ATTR(*namei*)
Implements `del TOS.name`, using *namei* as index into `co_names`.

STORE_GLOBAL(*namei*)
Works as *STORE_NAME*, but stores the name as a global.

DELETE_GLOBAL(*namei*)
Works as *DELETE_NAME*, but deletes a global name.

LOAD_CONST(*consti*)
Pushes `co_consts[consti]` onto the stack.

LOAD_NAME(*namei*)
Pushes the value associated with `co_names[namei]` onto the stack.

BUILD_TUPLE(*count*)
Creates a tuple consuming *count* items from the stack, and pushes the resulting tuple onto the stack.

BUILD_LIST(*count*)
Works as *BUILD_TUPLE*, but creates a list.

BUILD_SET(*count*)
Works as *BUILD_TUPLE*, but creates a set.

BUILD_MAP(*count*)
Pushes a new dictionary object onto the stack. Pops `2 * count` items so that the dictionary holds *count* entries: `{..., TOS3: TOS2, TOS1: TOS}`.

Alterado na versão 3.5: The dictionary is created from stack items instead of creating an empty dictionary pre-sized to hold *count* items.

BUILD_CONST_KEY_MAP(*count*)
The version of *BUILD_MAP* specialized for constant keys. Pops the top element on the stack which contains a tuple of keys, then starting from `TOS1`, pops *count* values to form values in the built dictionary.

Novo na versão 3.6.

BUILD_STRING(*count*)
Concatenates *count* strings from the stack and pushes the resulting string onto the stack.

Novo na versão 3.6.

BUILD_TUPLE_UNPACK(*count*)
Pops *count* iterables from the stack, joins them in a single tuple, and pushes the result. Implements iterable unpacking in tuple displays `(*x, *y, *z)`.

Novo na versão 3.5.

BUILD_TUPLE_UNPACK_WITH_CALL(*count*)
This is similar to *BUILD_TUPLE_UNPACK*, but is used for `f(*x, *y, *z)` call syntax. The stack item at position `count + 1` should be the corresponding callable `f`.

Novo na versão 3.6.

BUILD_LIST_UNPACK(*count*)
This is similar to *BUILD_TUPLE_UNPACK*, but pushes a list instead of tuple. Implements iterable unpacking in list displays `[*x, *y, *z]`.

Novo na versão 3.5.

BUILD_SET_UNPACK(*count*)
This is similar to *BUILD_TUPLE_UNPACK*, but pushes a set instead of tuple. Implements iterable unpacking in set displays `{*x, *y, *z}`.

Novo na versão 3.5.

BUILD_MAP_UNPACK(*count*)
Pops *count* mappings from the stack, merges them into a single dictionary, and pushes the result. Implements dictionary unpacking in dictionary displays `{**x, **y, **z}`.

Novo na versão 3.5.

BUILD_MAP_UNPACK_WITH_CALL(*count*)
This is similar to *BUILD_MAP_UNPACK*, but is used for `f(**x, **y, **z)` call syntax. The stack item at position `count + 2` should be the corresponding callable `f`.

Novo na versão 3.5.

Alterado na versão 3.6: The position of the callable is determined by adding 2 to the opcode argument instead of encoding it in the second byte of the argument.

LOAD_ATTR(*namei*)
Replaces TOS with `getattr(TOS, co_names[namei])`.

COMPARE_OP(*opname*)
Performs a Boolean operation. The operation name can be found in `cmp_op[opname]`.

IMPORT_NAME(*namei*)
Imports the module `co_names[namei]`. TOS and TOS1 are popped and provide the *fromlist* and *level* arguments of *__import__()*. The module object is pushed onto the stack. The current namespace is not affected: for a proper import statement, a subsequent *STORE_FAST* instruction modifies the namespace.

IMPORT_FROM(*namei*)
Loads the attribute `co_names[namei]` from the module found in TOS. The resulting object is pushed onto the stack, to be subsequently stored by a *STORE_FAST* instruction.

JUMP_FORWARD(*delta*)
Increments bytecode counter by *delta*.

POP_JUMP_IF_TRUE(*target*)
If TOS is true, sets the bytecode counter to *target*. TOS is popped.

Novo na versão 3.1.

POP_JUMP_IF_FALSE(*target*)
If TOS is false, sets the bytecode counter to *target*. TOS is popped.

Novo na versão 3.1.

JUMP_IF_TRUE_OR_POP(*target*)
If TOS is true, sets the bytecode counter to *target* and leaves TOS on the stack. Otherwise (TOS is false), TOS is popped.

Novo na versão 3.1.

JUMP_IF_FALSE_OR_POP(*target*)
If TOS is false, sets the bytecode counter to *target* and leaves TOS on the stack. Otherwise (TOS is true), TOS is popped.

Novo na versão 3.1.

JUMP_ABSOLUTE(*target*)
Set bytecode counter to *target*.

FOR_ITER (*delta*)
TOS is an *iterator*. Call its `__next__()` method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted TOS is popped, and the byte code counter is incremented by *delta*.

LOAD_GLOBAL (*namei*)
Loads the global named `co_names[namei]` onto the stack.

SETUP_LOOP (*delta*)
Pushes a block for a loop onto the block stack. The block spans from the current instruction with a size of *delta* bytes.

SETUP_EXCEPT (*delta*)
Pushes a try block from a try-except clause onto the block stack. *delta* points to the first except block.

SETUP_FINALLY (*delta*)
Pushes a try block from a try-except clause onto the block stack. *delta* points to the finally block.

LOAD_FAST (*var_num*)
Pushes a reference to the local `co_varnames[var_num]` onto the stack.

STORE_FAST (*var_num*)
Stores TOS into the local `co_varnames[var_num]`.

DELETE_FAST (*var_num*)
Deletes local `co_varnames[var_num]`.

LOAD_CLOSURE (*i*)
Pushes a reference to the cell contained in slot *i* of the cell and free variable storage. The name of the variable is `co_cellvars[i]` if *i* is less than the length of *co_cellvars*. Otherwise it is `co_freevars[i - len(co_cellvars)]`.

LOAD_DEREF (*i*)
Loads the cell contained in slot *i* of the cell and free variable storage. Pushes a reference to the object the cell contains on the stack.

LOAD_CLASSDEREF (*i*)
Much like *LOAD_DEREF* but first checks the locals dictionary before consulting the cell. This is used for loading free variables in class bodies.

Novo na versão 3.4.

STORE_DEREF (*i*)
Stores TOS into the cell contained in slot *i* of the cell and free variable storage.

DELETE_DEREF (*i*)
Empties the cell contained in slot *i* of the cell and free variable storage. Used by the `del` statement.

Novo na versão 3.2.

RAISE_VARARGS (*argc*)
Raises an exception using one of the 3 forms of the `raise` statement, depending on the value of *argc*:

- 0: `raise` (re-raise previous exception)
- 1: `raise TOS` (raise exception instance or type at `TOS`)
- 2: `raise TOS1 from TOS` (raise exception instance or type at `TOS1` with `__cause__` set to `TOS`)

CALL_FUNCTION (*argc*)
Calls a callable object with positional arguments. *argc* indicates the number of positional arguments. The top of the stack contains positional arguments, with the right-most argument on top. Below the arguments is a callable

object to call. CALL_FUNCTION pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Alterado na versão 3.6: This opcode is used only for calls with positional arguments.

CALL_FUNCTION_KW (*argc*)
Calls a callable object with positional (if any) and keyword arguments. *argc* indicates the total number of positional and keyword arguments. The top element on the stack contains a tuple of keyword argument names. Below that are keyword arguments in the order corresponding to the tuple. Below that are positional arguments, with the right-most parameter on top. Below the arguments is a callable object to call. CALL_FUNCTION_KW pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Alterado na versão 3.6: Keyword arguments are packed in a tuple instead of a dictionary, *argc* indicates the total number of arguments.

CALL_FUNCTION_EX (*flags*)
Calls a callable object with variable set of positional and keyword arguments. If the lowest bit of *flags* is set, the top of the stack contains a mapping object containing additional keyword arguments. Below that is an iterable object containing positional arguments and a callable object to call. *BUILD_MAP_UNPACK_WITH_CALL* and *BUILD_TUPLE_UNPACK_WITH_CALL* can be used for merging multiple mapping objects and iterables containing arguments. Before the callable is called, the mapping object and iterable object are each "unpacked" and their contents passed in as keyword and positional arguments respectively. CALL_FUNCTION_EX pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Novo na versão 3.6.

LOAD_METHOD (*namei*)
Loads a method named co_names[namei] from the TOS object. TOS is popped. This bytecode distinguishes two cases: if TOS has a method with the correct name, the bytecode pushes the unbound method and TOS. TOS will be used as the first argument (self) by *CALL_METHOD* when calling the unbound method. Otherwise, NULL and the object return by the attribute lookup are pushed.

Novo na versão 3.7.

CALL_METHOD (*argc*)
Calls a method. *argc* is the number of positional arguments. Keyword arguments are not supported. This opcode is designed to be used with *LOAD_METHOD*. Positional arguments are on top of the stack. Below them, the two items described in *LOAD_METHOD* are on the stack (either self and an unbound method object or NULL and an arbitrary callable). All of them are popped and the return value is pushed.

Novo na versão 3.7.

MAKE_FUNCTION (*flags*)
Pushes a new function object on the stack. From bottom to top, the consumed stack must consist of values if the argument carries a specified flag value

- 0x01 a tuple of default values for positional-only and positional-or-keyword parameters in positional order
- 0x02 a dictionary of keyword-only parameters' default values
- 0x04 an annotation dictionary
- 0x08 a tuple containing cells for free variables, making a closure
- the code associated with the function (at TOS1)
- the *qualified name* of the function (at TOS)

BUILD_SLICE (*argc*)
Pushes a slice object on the stack. *argc* must be 2 or 3. If it is 2, `slice(TOS1, TOS)` is pushed; if it is 3, `slice(TOS2, TOS1, TOS)` is pushed. See the *slice()* built-in function for more information.

EXTENDED_ARG (*ext*)
Prefixes any opcode which has an argument too big to fit into the default one byte. *ext* holds an additional byte which act as higher bits in the argument. For each opcode, at most three prefixal `EXTENDED_ARG` are allowed, forming an argument from two-byte to four-byte.

FORMAT_VALUE (*flags*)
Used for implementing formatted literal strings (f-strings). Pops an optional *fmt_spec* from the stack, then a required *value*. *flags* is interpreted as follows:

- `(flags & 0x03) == 0x00`: *value* is formatted as-is.
- `(flags & 0x03) == 0x01`: call *str()* on *value* before formatting it.
- `(flags & 0x03) == 0x02`: call *repr()* on *value* before formatting it.
- `(flags & 0x03) == 0x03`: call *ascii()* on *value* before formatting it.
- `(flags & 0x04) == 0x04`: pop *fmt_spec* from the stack and use it, else use an empty *fmt_spec*.

Formatting is performed using `PyObject_Format()`. The result is pushed on the stack.

Novo na versão 3.6.

HAVE_ARGUMENT
This is not really an opcode. It identifies the dividing line between opcodes which don't use their argument and those that do (`< HAVE_ARGUMENT` and `>= HAVE_ARGUMENT`, respectively).

Alterado na versão 3.6: Now every instruction has an argument, but opcodes `< HAVE_ARGUMENT` ignore it. Before, only opcodes `>= HAVE_ARGUMENT` had an argument.

33.12.4 Opcode collections

These collections are provided for automatic introspection of bytecode instructions:

`dis.`**opname**
Sequence of operation names, indexable using the bytecode.

`dis.`**opmap**
Dictionary mapping operation names to bytecodes.

`dis.`**cmp_op**
Sequence of all compare operation names.

`dis.`**hasconst**
Sequence of bytecodes that access a constant.

`dis.`**hasfree**
Sequence of bytecodes that access a free variable (note that 'free' in this context refers to names in the current scope that are referenced by inner scopes or names in outer scopes that are referenced from this scope. It does *not* include references to global or builtin scopes).

`dis.`**hasname**
Sequence of bytecodes that access an attribute by name.

`dis.`**hasjrel**
Sequence of bytecodes that have a relative jump target.

dis.**hasjabs**
Sequence of bytecodes that have an absolute jump target.

dis.**haslocal**
Sequence of bytecodes that access a local variable.

dis.**hascompare**
Sequence of bytecodes of Boolean operations.

33.13 pickletools — Ferramentas para desenvolvedores pickle

Código Fonte: Lib/pickletools.py

Este módulo contém várias constantes relacionadas aos detalhes íntimos do módulo *pickle*, alguns comentários extensos sobre a implementação e algumas funções úteis para analisar dados em conserva. O conteúdo deste módulo é útil para desenvolvedores do núcleo Python que estão trabalhando no *pickle*; usuários comuns do módulo *pickle* provavelmente não acharão o módulo *pickletools* relevante.

33.13.1 Uso na linha de comando

Novo na versão 3.2.

Quando chamado a partir da linha de comando, `python -m pickletools` irá desmontar o conteúdo de um ou mais arquivos pickle. Note que se você quiser ver o objeto Python armazenado no pickle ao invés dos detalhes do formato pickle, você pode usar `-m pickle`. No entanto, quando o arquivo pickle que você deseja examinar vem de uma fonte não confiável, `-m pickletools` é uma opção mais segura porque não executa bytecode pickle.

Por exemplo, com uma tupla `(1, 2)` tratada com pickling no arquivo `x.pickle`:

```
$ python -m pickle x.pickle
(1, 2)

$ python -m pickletools x.pickle
    0: \x80 PROTO      3
    2: K    BININT1    1
    4: K    BININT1    2
    6: \x86 TUPLE2
    7: q    BINPUT     0
    9: .    STOP
highest protocol among opcodes = 2
```

Opções da linha de comando

-a, --annotate
Anota cada linha com uma descrição curta do código de operação.

-o, --output=<file>
Nome de um arquivo no qual a saída deve ser escrita.

-l, --indentlevel=<num>
O número de espaços em branco para indentar um novo nível MARK.

-m, --memo
Quando vários objetos são desmontados, preserva memo entre as desmontagens.

-p, --preamble=<preamble>
Quando mais de um arquivo pickle for especificado, imprime o preâmbulo fornecido antes de cada desmontagem.

33.13.2 Interface programática

pickletools.**dis**(*pickle*, *out=None*, *memo=None*, *indentlevel=4*, *annotate=0*)
Produz uma desmontagem simbólica do pickle para o objeto arquivo ou similar *out*, tendo como padrão `sys.stdout`. *pickle* pode ser uma string ou um objeto arquivo ou similar. *memo* pode ser um dicionário Python que será usado como memo do pickle; ele pode ser usado para realizar desmontagens em vários pickles criados pelo mesmo pickler. Níveis sucessivos, indicados por códigos de operação `MARK` no fluxo, são indentados por espaços *indentlevel*. Se um valor diferente de zero for fornecido para *annotate*, cada código de operação na saída será anotado com uma breve descrição. O valor de *annotate* é usado como uma dica para a coluna onde a anotação deve começar.

Novo na versão 3.2: O argumento *annotate*.

pickletools.**genops**(*pickle*)
Fornece um *iterador* sobre todos os códigos de operação em um pickle, retornando uma sequência de triplos `(opcode, arg, pos)`. *opcode* é uma instância de uma classe `OpcodeInfo`; *arg* é o valor decodificado, como um objeto Python, do argumento do opcode; *pos* é a posição em que este código de operação está localizado. *pickle* pode ser uma string ou um objeto arquivo ou similar.

pickletools.**optimize**(*picklestring*)
Retorna uma nova string pickle equivalente após eliminar códigos de operação `PUT` não utilizados. O pickle otimizado é mais curto, leva menos tempo de transmissão, requer menos espaço de armazenamento e efetua unpickling com mais eficiência.

CAPÍTULO 34

Serviços Diversos

Os módulos descritos neste capítulo fornecem diversos serviços que estão disponíveis em todas as versões do Python. Aqui temos uma visão geral:

34.1 `formatter` — Formatação de saída genérica

Obsoleto desde a versão 3.4: Due to lack of usage, the formatter module has been deprecated.

This module supports two interface definitions, each with multiple implementations: The *formatter* interface, and the *writer* interface which is required by the formatter interface.

Formatter objects transform an abstract flow of formatting events into specific output events on writer objects. Formatters manage several stack structures to allow various properties of a writer object to be changed and restored; writers need not be able to handle relative changes nor any sort of "change back" operation. Specific writer properties which may be controlled via formatter objects are horizontal alignment, font, and left margin indentations. A mechanism is provided which supports providing arbitrary, non-exclusive style settings to a writer as well. Additional interfaces facilitate formatting events which are not reversible, such as paragraph separation.

Writer objects encapsulate device interfaces. Abstract devices, such as file formats, are supported as well as physical devices. The provided implementations all work with abstract devices. The interface makes available mechanisms for setting the properties which formatter objects manage and inserting data into the output.

34.1.1 The Formatter Interface

Interfaces to create formatters are dependent on the specific formatter class being instantiated. The interfaces described below are the required interfaces which all formatters must support once initialized.

One data element is defined at the module level:

`formatter.`**`AS_IS`**
Value which can be used in the font specification passed to the `push_font()` method described below, or as the new value to any other `push_property()` method. Pushing the `AS_IS` value allows the corresponding `pop_property()` method to be called without having to track whether the property was changed.

The following attributes are defined for formatter instance objects:

`formatter.`**`writer`**
The writer instance with which the formatter interacts.

`formatter.`**`end_paragraph`**(*blanklines*)
Close any open paragraphs and insert at least *blanklines* before the next paragraph.

`formatter.`**`add_line_break`**()
Add a hard line break if one does not already exist. This does not break the logical paragraph.

`formatter.`**`add_hor_rule`**(**args*, ***kw*)
Insert a horizontal rule in the output. A hard break is inserted if there is data in the current paragraph, but the logical paragraph is not broken. The arguments and keywords are passed on to the writer's `send_line_break()` method.

`formatter.`**`add_flowing_data`**(*data*)
Provide data which should be formatted with collapsed whitespace. Whitespace from preceding and successive calls to *`add_flowing_data()`* is considered as well when the whitespace collapse is performed. The data which is passed to this method is expected to be word-wrapped by the output device. Note that any word-wrapping still must be performed by the writer object due to the need to rely on device and font information.

`formatter.`**`add_literal_data`**(*data*)
Provide data which should be passed to the writer unchanged. Whitespace, including newline and tab characters, are considered legal in the value of *data*.

`formatter.`**`add_label_data`**(*format*, *counter*)
Insert a label which should be placed to the left of the current left margin. This should be used for constructing bulleted or numbered lists. If the *format* value is a string, it is interpreted as a format specification for *counter*, which should be an integer. The result of this formatting becomes the value of the label; if *format* is not a string it is used as the label value directly. The label value is passed as the only argument to the writer's `send_label_data()` method. Interpretation of non-string label values is dependent on the associated writer.

Format specifications are strings which, in combination with a counter value, are used to compute label values. Each character in the format string is copied to the label value, with some characters recognized to indicate a transform on the counter value. Specifically, the character `'1'` represents the counter value formatter as an Arabic number, the characters `'A'` and `'a'` represent alphabetic representations of the counter value in upper and lower case, respectively, and `'I'` and `'i'` represent the counter value in Roman numerals, in upper and lower case. Note that the alphabetic and roman transforms require that the counter value be greater than zero.

`formatter.`**`flush_softspace`**()
Send any pending whitespace buffered from a previous call to *`add_flowing_data()`* to the associated writer object. This should be called before any direct manipulation of the writer object.

`formatter.`**`push_alignment`**(*align*)
Push a new alignment setting onto the alignment stack. This may be *`AS_IS`* if no change is desired. If the alignment value is changed from the previous setting, the writer's `new_alignment()` method is called with the *align* value.

formatter.**pop_alignment**()
Restaurar o anterior

formatter.**push_font**(*(size, italic, bold, teletype)*)
Change some or all font properties of the writer object. Properties which are not set to *AS_IS* are set to the values passed in while others are maintained at their current settings. The writer's `new_font()` method is called with the fully resolved font specification.

formatter.**pop_font**()
Restaurar a fonte anterior.

formatter.**push_margin**(*margin*)
Increase the number of left margin indentations by one, associating the logical tag *margin* with the new indentation. The initial margin level is `0`. Changed values of the logical tag must be true values; false values other than *AS_IS* are not sufficient to change the margin.

formatter.**pop_margin**()
Restaurar a margem anterior.

formatter.**push_style**(**styles*)
Push any number of arbitrary style specifications. All styles are pushed onto the styles stack in order. A tuple representing the entire stack, including *AS_IS* values, is passed to the writer's `new_styles()` method.

formatter.**pop_style**(*n=1*)
Pop the last *n* style specifications passed to *push_style()*. A tuple representing the revised stack, including *AS_IS* values, is passed to the writer's `new_styles()` method.

formatter.**set_spacing**(*spacing*)
Set the spacing style for the writer.

formatter.**assert_line_data**(*flag=1*)
Inform the formatter that data has been added to the current paragraph out-of-band. This should be used when the writer has been manipulated directly. The optional *flag* argument can be set to false if the writer manipulations produced a hard line break at the end of the output.

34.1.2 Formatter Implementations

Two implementations of formatter objects are provided by this module. Most applications may use one of these classes without modification or subclassing.

class formatter.**NullFormatter**(*writer=None*)
A formatter which does nothing. If *writer* is omitted, a *NullWriter* instance is created. No methods of the writer are called by *NullFormatter* instances. Implementations should inherit from this class if implementing a writer interface but don't need to inherit any implementation.

class formatter.**AbstractFormatter**(*writer*)
The standard formatter. This implementation has demonstrated wide applicability to many writers, and may be used directly in most circumstances. It has been used to implement a full-featured World Wide Web browser.

34.1.3 The Writer Interface

Interfaces to create writers are dependent on the specific writer class being instantiated. The interfaces described below are the required interfaces which all writers must support once initialized. Note that while most applications can use the *AbstractFormatter* class as a formatter, the writer must typically be provided by the application.

writer.**flush**()
: Flush any buffered output or device control events.

writer.**new_alignment**(*align*)
: Set the alignment style. The *align* value can be any object, but by convention is a string or `None`, where `None` indicates that the writer's "preferred" alignment should be used. Conventional *align* values are `'left'`, `'center'`, `'right'`, and `'justify'`.

writer.**new_font**(*font*)
: Set the font style. The value of *font* will be `None`, indicating that the device's default font should be used, or a tuple of the form `(size, italic, bold, teletype)`. Size will be a string indicating the size of font that should be used; specific strings and their interpretation must be defined by the application. The *italic*, *bold*, and *teletype* values are Boolean values specifying which of those font attributes should be used.

writer.**new_margin**(*margin*, *level*)
: Set the margin level to the integer *level* and the logical tag to *margin*. Interpretation of the logical tag is at the writer's discretion; the only restriction on the value of the logical tag is that it not be a false value for non-zero values of *level*.

writer.**new_spacing**(*spacing*)
: Set the spacing style to *spacing*.

writer.**new_styles**(*styles*)
: Set additional styles. The *styles* value is a tuple of arbitrary values; the value *AS_IS* should be ignored. The *styles* tuple may be interpreted either as a set or as a stack depending on the requirements of the application and writer implementation.

writer.**send_line_break**()
: Quebra a linha atual.

writer.**send_paragraph**(*blankline*)
: Produce a paragraph separation of at least *blankline* blank lines, or the equivalent. The *blankline* value will be an integer. Note that the implementation will receive a call to *send_line_break()* before this call if a line break is needed; this method should not include ending the last line of the paragraph. It is only responsible for vertical spacing between paragraphs.

writer.**send_hor_rule**(**args*, ***kw*)
: Display a horizontal rule on the output device. The arguments to this method are entirely application- and writer-specific, and should be interpreted with care. The method implementation may assume that a line break has already been issued via *send_line_break()*.

writer.**send_flowing_data**(*data*)
: Output character data which may be word-wrapped and re-flowed as needed. Within any sequence of calls to this method, the writer may assume that spans of multiple whitespace characters have been collapsed to single space characters.

writer.**send_literal_data**(*data*)
: Output character data which has already been formatted for display. Generally, this should be interpreted to mean that line breaks indicated by newline characters should be preserved and no new line breaks should be introduced. The data may contain embedded newline and tab characters, unlike data provided to the `send_formatted_data()` interface.

writer.**send_label_data**(*data*)
: Set *data* to the left of the current left margin, if possible. The value of *data* is not restricted; treatment of non-string

values is entirely application- and writer-dependent. This method will only be called at the beginning of a line.

34.1.4 Implementações de Writer

Three implementations of the writer object interface are provided as examples by this module. Most applications will need to derive new writer classes from the `NullWriter` class.

class formatter.**NullWriter**
A writer which only provides the interface definition; no actions are taken on any methods. This should be the base class for all writers which do not need to inherit any implementation methods.

class formatter.**AbstractWriter**
A writer which can be used in debugging formatters, but not much else. Each method simply announces itself by printing its name and arguments on standard output.

class formatter.**DumbWriter**(*file=None*, *maxcol=72*)
Simple writer class which writes output on the *file object* passed in as *file* or, if *file* is omitted, on standard output. The output is simply word-wrapped to the number of columns specified by *maxcol*. This class is suitable for reflowing a sequence of paragraphs.

CAPÍTULO 35

Serviços Específicos do MS Windows

Este capítulo descreve módulos que estão disponíveis somente na plataformas MS Windows.

35.1 `msilib` — Read and write Microsoft Installer files

Código Fonte: Lib/msilib/__init__.py

The *msilib* supports the creation of Microsoft Installer (`.msi`) files. Because these files often contain an embedded "cabinet" file (`.cab`), it also exposes an API to create CAB files. Support for reading `.cab` files is currently not implemented; read support for the `.msi` database is possible.

This package aims to provide complete access to all tables in an `.msi` file, therefore, it is a fairly low-level API. Two primary applications of this package are the *distutils* command `bdist_msi`, and the creation of Python installer package itself (although that currently uses a different version of `msilib`).

The package contents can be roughly split into four parts: low-level CAB routines, low-level MSI routines, higher-level MSI routines, and standard table structures.

`msilib.`**`FCICreate`**(*cabname*, *files*)
Create a new CAB file named *cabname*. *files* must be a list of tuples, each containing the name of the file on disk, and the name of the file inside the CAB file.

The files are added to the CAB file in the order they appear in the list. All files are added into a single CAB file, using the MSZIP compression algorithm.

Callbacks to Python for the various steps of MSI creation are currently not exposed.

`msilib.`**`UuidCreate`**()
Return the string representation of a new unique identifier. This wraps the Windows API functions `UuidCreate()` and `UuidToString()`.

`msilib.`**`OpenDatabase`**(*path*, *persist*)
Return a new database object by calling MsiOpenDatabase. *path* is the file name of the MSI file; *persist* can be one of the constants `MSIDBOPEN_CREATEDIRECT`, `MSIDBOPEN_CREATE`, `MSIDBOPEN_DIRECT`,

MSIDBOPEN_READONLY, or MSIDBOPEN_TRANSACT, and may include the flag MSIDBOPEN_PATCHFILE. See the Microsoft documentation for the meaning of these flags; depending on the flags, an existing database is opened, or a new one created.

msilib.**CreateRecord**(*count*)
Return a new record object by calling MSICreateRecord(). *count* is the number of fields of the record.

msilib.**init_database**(*name*, *schema*, *ProductName*, *ProductCode*, *ProductVersion*, *Manufacturer*)
Create and return a new database *name*, initialize it with *schema*, and set the properties *ProductName*, *ProductCode*, *ProductVersion*, and *Manufacturer*.

schema must be a module object containing tables and _Validation_records attributes; typically, *msilib.schema* should be used.

The database will contain just the schema and the validation records when this function returns.

msilib.**add_data**(*database*, *table*, *records*)
Add all *records* to the table named *table* in *database*.

The *table* argument must be one of the predefined tables in the MSI schema, e.g. 'Feature', 'File', 'Component', 'Dialog', 'Control', etc.

records should be a list of tuples, each one containing all fields of a record according to the schema of the table. For optional fields, None can be passed.

Field values can be ints, strings, or instances of the Binary class.

class msilib.**Binary**(*filename*)
Represents entries in the Binary table; inserting such an object using *add_data()* reads the file named *filename* into the table.

msilib.**add_tables**(*database*, *module*)
Add all table content from *module* to *database*. *module* must contain an attribute *tables* listing all tables for which content should be added, and one attribute per table that has the actual content.

This is typically used to install the sequence tables.

msilib.**add_stream**(*database*, *name*, *path*)
Add the file *path* into the _Stream table of *database*, with the stream name *name*.

msilib.**gen_uuid**()
Return a new UUID, in the format that MSI typically requires (i.e. in curly braces, and with all hexdigits in uppercase).

Ver também:

FCICreate UuidCreate UuidToString

35.1.1 Objetos de banco de dados.

Database.**OpenView**(*sql*)
Return a view object, by calling MSIDatabaseOpenView(). *sql* is the SQL statement to execute.

Database.**Commit**()
Commit the changes pending in the current transaction, by calling MSIDatabaseCommit().

Database.**GetSummaryInformation**(*count*)
Return a new summary information object, by calling MsiGetSummaryInformation(). *count* is the maximum number of updated values.

Database.**Close**()
Close the database object, through MsiCloseHandle().

Novo na versão 3.7.

Ver também:

MSIDatabaseOpenView MSIDatabaseCommit MSIGetSummaryInformation MsiCloseHandle

35.1.2 View Objects

View.**Execute**(*params*)
Execute the SQL query of the view, through MSIViewExecute(). If *params* is not None, it is a record describing actual values of the parameter tokens in the query.

View.**GetColumnInfo**(*kind*)
Return a record describing the columns of the view, through calling MsiViewGetColumnInfo(). *kind* can be either MSICOLINFO_NAMES or MSICOLINFO_TYPES.

View.**Fetch**()
Return a result record of the query, through calling MsiViewFetch().

View.**Modify**(*kind*, *data*)
Modify the view, by calling MsiViewModify(). *kind* can be one of MSIMODIFY_SEEK, MSIMODIFY_REFRESH, MSIMODIFY_INSERT, MSIMODIFY_UPDATE, MSIMODIFY_ASSIGN, MSIMODIFY_REPLACE, MSIMODIFY_MERGE, MSIMODIFY_DELETE, MSIMODIFY_INSERT_TEMPORARY, MSIMODIFY_VALIDATE, MSIMODIFY_VALIDATE_NEW, MSIMODIFY_VALIDATE_FIELD, or MSIMODIFY_VALIDATE_DELETE.

data must be a record describing the new data.

View.**Close**()
Close the view, through MsiViewClose().

Ver também:

MsiViewExecute MSIViewGetColumnInfo MsiViewFetch MsiViewModify MsiViewClose

35.1.3 Summary Information Objects

SummaryInformation.**GetProperty**(*field*)
Return a property of the summary, through MsiSummaryInfoGetProperty(). *field* is the name of the property, and can be one of the constants PID_CODEPAGE, PID_TITLE, PID_SUBJECT, PID_AUTHOR, PID_KEYWORDS, PID_COMMENTS, PID_TEMPLATE, PID_LASTAUTHOR, PID_REVNUMBER, PID_LASTPRINTED, PID_CREATE_DTM, PID_LASTSAVE_DTM, PID_PAGECOUNT, PID_WORDCOUNT, PID_CHARCOUNT, PID_APPNAME, or PID_SECURITY.

SummaryInformation.**GetPropertyCount**()
Return the number of summary properties, through MsiSummaryInfoGetPropertyCount().

SummaryInformation.**SetProperty**(*field*, *value*)
Set a property through MsiSummaryInfoSetProperty(). *field* can have the same values as in *GetProperty()*, *value* is the new value of the property. Possible value types are integer and string.

SummaryInformation.**Persist**()
Write the modified properties to the summary information stream, using MsiSummaryInfoPersist().

Ver também:

MsiSummaryInfoGetProperty MsiSummaryInfoGetPropertyCount MsiSummaryInfoSetProperty MsiSummaryInfoPersist

35.1.4 Record Objects

`Record.`**`GetFieldCount`**`()`
Return the number of fields of the record, through `MsiRecordGetFieldCount()`.

`Record.`**`GetInteger`**`(`*field*`)`
Return the value of *field* as an integer where possible. *field* must be an integer.

`Record.`**`GetString`**`(`*field*`)`
Return the value of *field* as a string where possible. *field* must be an integer.

`Record.`**`SetString`**`(`*field*, *value*`)`
Set *field* to *value* through `MsiRecordSetString()`. *field* must be an integer; *value* a string.

`Record.`**`SetStream`**`(`*field*, *value*`)`
Set *field* to the contents of the file named *value*, through `MsiRecordSetStream()`. *field* must be an integer; *value* a string.

`Record.`**`SetInteger`**`(`*field*, *value*`)`
Set *field* to *value* through `MsiRecordSetInteger()`. Both *field* and *value* must be an integer.

`Record.`**`ClearData`**`()`
Set all fields of the record to 0, through `MsiRecordClearData()`.

Ver também:

MsiRecordGetFieldCount MsiRecordSetString MsiRecordSetStream MsiRecordSetInteger MsiRecordClearData

35.1.5 Erros

All wrappers around MSI functions raise `MSIError`; the string inside the exception will contain more detail.

35.1.6 CAB Objects

`class` `msilib.`**`CAB`**`(`*name*`)`
The class *CAB* represents a CAB file. During MSI construction, files will be added simultaneously to the `Files` table, and to a CAB file. Then, when all files have been added, the CAB file can be written, then added to the MSI file.

name is the name of the CAB file in the MSI file.

`append``(`*full*, *file*, *logical*`)`
Add the file with the pathname *full* to the CAB file, under the name *logical*. If there is already a file named *logical*, a new file name is created.

Return the index of the file in the CAB file, and the new name of the file inside the CAB file.

`commit``(`*database*`)`
Generate a CAB file, add it as a stream to the MSI file, put it into the `Media` table, and remove the generated file from the disk.

35.1.7 Objetos Directory

class msilib.**Directory**(*database*, *cab*, *basedir*, *physical*, *logical*, *default*[, *componentflags*])
Create a new directory in the Directory table. There is a current component at each point in time for the directory, which is either explicitly created through *start_component()*, or implicitly when files are added for the first time. Files are added into the current component, and into the cab file. To create a directory, a base directory object needs to be specified (can be None), the path to the physical directory, and a logical directory name. *default* specifies the DefaultDir slot in the directory table. *componentflags* specifies the default flags that new components get.

start_component(*component=None*, *feature=None*, *flags=None*, *keyfile=None*, *uuid=None*)
Add an entry to the Component table, and make this component the current component for this directory. If no component name is given, the directory name is used. If no *feature* is given, the current feature is used. If no *flags* are given, the directory's default flags are used. If no *keyfile* is given, the KeyPath is left null in the Component table.

add_file(*file*, *src=None*, *version=None*, *language=None*)
Add a file to the current component of the directory, starting a new one if there is no current component. By default, the file name in the source and the file table will be identical. If the *src* file is specified, it is interpreted relative to the current directory. Optionally, a *version* and a *language* can be specified for the entry in the File table.

glob(*pattern*, *exclude=None*)
Add a list of files to the current component as specified in the glob pattern. Individual files can be excluded in the *exclude* list.

remove_pyc()
Remove .pyc files on uninstall.

Ver também:

Directory Table File Table Component Table FeatureComponents Table

35.1.8 Recursos

class msilib.**Feature**(*db*, *id*, *title*, *desc*, *display*, *level=1*, *parent=None*, *directory=None*, *attributes=0*)
Add a new record to the Feature table, using the values *id*, *parent.id*, *title*, *desc*, *display*, *level*, *directory*, and *attributes*. The resulting feature object can be passed to the start_component() method of *Directory*.

set_current()
Make this feature the current feature of *msilib*. New components are automatically added to the default feature, unless a feature is explicitly specified.

Ver também:

Feature Table

35.1.9 Classes GUI

msilib provides several classes that wrap the GUI tables in an MSI database. However, no standard user interface is provided; use `bdist_msi` to create MSI files with a user-interface for installing Python packages.

class msilib.**Control**(*dlg*, *name*)
Base class of the dialog controls. *dlg* is the dialog object the control belongs to, and *name* is the control's name.

event(*event*, *argument*, *condition=1*, *ordering=None*)
Make an entry into the `ControlEvent` table for this control.

mapping(*event*, *attribute*)
Make an entry into the `EventMapping` table for this control.

condition(*action*, *condition*)
Cria uma entrada na tabela `ControlCondition` para este controle.

class msilib.**RadioButtonGroup**(*dlg*, *name*, *property*)
Create a radio button control named *name*. *property* is the installer property that gets set when a radio button is selected.

add(*name*, *x*, *y*, *width*, *height*, *text*, *value=None*)
Add a radio button named *name* to the group, at the coordinates *x*, *y*, *width*, *height*, and with the label *text*. If *value* is `None`, it defaults to *name*.

class msilib.**Dialog**(*db*, *name*, *x*, *y*, *w*, *h*, *attr*, *title*, *first*, *default*, *cancel*)
Return a new *Dialog* object. An entry in the `Dialog` table is made, with the specified coordinates, dialog attributes, title, name of the first, default, and cancel controls.

control(*name*, *type*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *control_next*, *help*)
Return a new *Control* object. An entry in the `Control` table is made with the specified parameters.

This is a generic method; for specific types, specialized methods are provided.

text(*name*, *x*, *y*, *width*, *height*, *attributes*, *text*)
Add and return a `Text` control.

bitmap(*name*, *x*, *y*, *width*, *height*, *text*)
Add and return a `Bitmap` control.

line(*name*, *x*, *y*, *width*, *height*)
Add and return a `Line` control.

pushbutton(*name*, *x*, *y*, *width*, *height*, *attributes*, *text*, *next_control*)
Add and return a `PushButton` control.

radiogroup(*name*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *next_control*)
Add and return a `RadioButtonGroup` control.

checkbox(*name*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *next_control*)
Add and return a `CheckBox` control.

Ver também:

Dialog Table Control Table Control Types ControlCondition Table ControlEvent Table EventMapping Table RadioButton Table

35.1.10 Precomputed tables

msilib provides a few subpackages that contain only schema and table definitions. Currently, these definitions are based on MSI version 2.0.

msilib.**schema**
: This is the standard MSI schema for MSI 2.0, with the *tables* variable providing a list of table definitions, and *_Validation_records* providing the data for MSI validation.

msilib.**sequence**
: This module contains table contents for the standard sequence tables: *AdminExecuteSequence*, *AdminUISequence*, *AdvtExecuteSequence*, *InstallExecuteSequence*, and *InstallUISequence*.

msilib.**text**
: This module contains definitions for the UIText and ActionText tables, for the standard installer actions.

35.2 msvcrt — Rotinas úteis do tempo de execução do MS VC++

Essas funções fornecem acesso a alguns recursos úteis nas plataformas Windows. Alguns módulos de nível superior usam essas funções para criar as implementações do Windows de seus serviços. Por exemplo, o módulo *getpass* usa isso na implementação da função *getpass()*.

Mais documentação sobre essas funções pode ser encontrada na documentação da API da plataforma.

O módulo implementa as variantes normal e ampla de caracteres da API de E/S do console. A API normal lida apenas com caracteres ASCII e é de uso limitado para aplicativos internacionalizados. A API ampla de caracteres deve ser usada sempre que possível.

Alterado na versão 3.3: As operações neste módulo agora levantam *OSError* onde *IOError* foi levantado.

35.2.1 Operações com arquivos

msvcrt.**locking**(*fd*, *mode*, *nbytes*)
: Bloqueia parte de um arquivo com base no descritor de arquivo *fd* no tempo de execução C. Levanta *OSError* em falha. A região bloqueada do arquivo se estende da posição atual do arquivo para *nbytes* bytes e pode continuar além do final do arquivo. *mode* deve ser uma das constantes `LK_*` listadas abaixo. Várias regiões em um arquivo podem estar bloqueadas ao mesmo tempo, mas não podem se sobrepor. Regiões adjacentes não são mescladas; eles devem ser desbloqueados individualmente.

msvcrt.**LK_LOCK**
msvcrt.**LK_RLCK**
: Bloqueia os bytes especificados. Se os bytes não puderem ser bloqueados, o programa tentará imediatamente novamente após 1 segundo. Se, após 10 tentativas, os bytes não puderem ser bloqueados, *OSError* será levantado.

msvcrt.**LK_NBLCK**
msvcrt.**LK_NBRLCK**
: Bloqueia os bytes especificados. Se os bytes não puderem ser bloqueados, *OSError* é levantado.

msvcrt.**LK_UNLCK**
: Desbloqueia os bytes especificados, que devem ter sido bloqueados anteriormente.

msvcrt.**setmode**(*fd*, *flags*)
: Defina o modo de conversão de final de linha para o descritor de arquivo *fd*. Para configurá-lo no modo de texto, *flags* deve ser *os.O_TEXT*; para binário, deve ser *os.O_BINARY*.

msvcrt.**open_osfhandle**(*handle*, *flags*)
Cria um descritor de arquivo em tempo de execução C a partir do identificador de arquivo *handle*. O parâmetro *flags* deve ser um OR bit a bit de *os.O_APPEND*, *os.O_RDONLY* e *os.O_TEXT*. O descritor de arquivo retornado pode ser usado como um parâmetro para *os.fdopen()* para criar um objeto de arquivo.

msvcrt.**get_osfhandle**(*fd*)
Retorna o identificador de arquivo para o descritor de arquivo *fd*. Leva *OSError* se *fd* não for reconhecido.

35.2.2 E/S de console

msvcrt.**kbhit**()
Retorna `True` se um pressionamento de tecla estiver aguardando para ser lido.

msvcrt.**getch**()
Lê um pressionamento de tecla e retorna o caractere resultante como uma sequência de bytes. Nada é ecoado no console. Essa chamada será bloqueada se um pressionamento de tecla ainda não estiver disponível, mas não esperará que `Enter` seja pressionado. Se a tecla pressionada for uma tecla de função especial, ela retornará `\000'` ou `'\xe0'`; a próxima chamada retornará o código da chave. A tecla `Control-C` não pode ser lida com esta função.

msvcrt.**getwch**()
Variante com caractere largo de *getch()*, retornando um valor Unicode.

msvcrt.**getche**()
Semelhante a *getch()*, mas o pressionamento de tecla será repetido se representar um caractere imprimível.

msvcrt.**getwche**()
Variante com caractere largo de *getche()*, retornando um valor Unicode.

msvcrt.**putch**(*char*)
Imprime a sequência de bytes *char* no console sem armazenar em buffer.

msvcrt.**putwch**(*unicode_char*)
Variante com caractere largo de *putch()*, retornando um valor Unicode.

msvcrt.**ungetch**(*char*)
Faz com que a string de bytes *char* seja "empurrada" para o buffer do console; será o próximo caractere lido por *getch()* ou *getche()*.

msvcrt.**ungetwch**(*unicode_char*)
Variante com caractere largo de *ungetch()*, retornando um valor Unicode.

35.2.3 Outras funções

msvcrt.**heapmin**()
Força o heap `malloc()` a ser limpado e retorna os blocos não utilizados ao sistema operacional. Em caso de falha, isso gera *OSError*.

35.3 winreg – Registro de acesso do Windows

These functions expose the Windows registry API to Python. Instead of using an integer as the registry handle, a *handle object* is used to ensure that the handles are closed correctly, even if the programmer neglects to explicitly close them.

Alterado na versão 3.3: Várias funções neste módulo utilizadas para levantar um:exc:*WindowsError*, que agora é um pseudônimo de:exc:*OSError*.

35.3.1 Funções

This module offers the following functions:

winreg.**CloseKey**(*hkey*)
Closes a previously opened registry key. The *hkey* argument specifies a previously opened key.

> **Nota:** If *hkey* is not closed using this method (or via `hkey.Close()`), it is closed when the *hkey* object is destroyed by Python.

winreg.**ConnectRegistry**(*computer_name*, *key*)
Establishes a connection to a predefined registry handle on another computer, and returns a *handle object*.

computer_name is the name of the remote computer, of the form `r"\\computername"`. If `None`, the local computer is used.

key is the predefined handle to connect to.

The return value is the handle of the opened key. If the function fails, an `OSError` exception is raised.

Alterado na versão 3.3: See *above*.

winreg.**CreateKey**(*key*, *sub_key*)
Creates or opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the key this method opens or creates.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an `OSError` exception is raised.

Alterado na versão 3.3: See *above*.

winreg.**CreateKeyEx**(*key*, *sub_key*, *reserved=0*, *access=KEY_WRITE*)
Creates or opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the key this method opens or creates.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is `KEY_WRITE`. See *Access Rights* for other allowed values.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an *OSError* exception is raised.

Novo na versão 3.2.

Alterado na versão 3.3: See *above*.

winreg.**DeleteKey**(*key*, *sub_key*)
Exclui a chave especificada.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an *OSError* exception is raised.

Alterado na versão 3.3: See *above*.

winreg.**DeleteKeyEx**(*key*, *sub_key*, *access=KEY_WOW64_64KEY*, *reserved=0*)
Exclui a chave especificada.

Nota: The *DeleteKeyEx()* function is implemented with the RegDeleteKeyEx Windows API function, which is specific to 64-bit versions of Windows. See the RegDeleteKeyEx documentation.

---

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is *KEY_WOW64_64KEY*. See *Access Rights* for other allowed values.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an *OSError* exception is raised.

On unsupported Windows versions, *NotImplementedError* is raised.

Novo na versão 3.2.

Alterado na versão 3.3: See *above*.

winreg.**DeleteValue**(*key*, *value*)
Removes a named value from a registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value is a string that identifies the value to remove.

winreg.**EnumKey**(*key*, *index*)
Enumerates subkeys of an open registry key, returning a string.

key is an already open key, or one of the predefined *HKEY_* constants*.

index is an integer that identifies the index of the key to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly until an `OSError` exception is raised, indicating, no more values are available.

Alterado na versão 3.3: See *above*.

`winreg.`**`EnumValue`**(*key*, *index*)
Enumerates values of an open registry key, returning a tuple.

key is an already open key, or one of the predefined *HKEY_* constants*.

index is an integer that identifies the index of the value to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly, until an `OSError` exception is raised, indicating no more values.

The result is a tuple of 3 items:

Index	Significado
0	A string that identifies the value name
1	An object that holds the value data, and whose type depends on the underlying registry type
2	An integer that identifies the type of the value data (see table in docs for `SetValueEx()`)

Alterado na versão 3.3: See *above*.

`winreg.`**`ExpandEnvironmentStrings`**(*str*)
Expands environment variable placeholders `%NAME%` in strings like `REG_EXPAND_SZ`:

```
>>> ExpandEnvironmentStrings('%windir%')
'C:\\Windows'
```

`winreg.`**`FlushKey`**(*key*)
Writes all the attributes of a key to the registry.

key is an already open key, or one of the predefined *HKEY_* constants*.

It is not necessary to call `FlushKey()` to change a key. Registry changes are flushed to disk by the registry using its lazy flusher. Registry changes are also flushed to disk at system shutdown. Unlike `CloseKey()`, the `FlushKey()` method returns only when all the data has been written to the registry. An application should only call `FlushKey()` if it requires absolute certainty that registry changes are on disk.

Nota: If you don't know whether a `FlushKey()` call is required, it probably isn't.

`winreg.`**`LoadKey`**(*key*, *sub_key*, *file_name*)
Creates a subkey under the specified key and stores registration information from a specified file into that subkey.

key is a handle returned by `ConnectRegistry()` or one of the constants `HKEY_USERS` or `HKEY_LOCAL_MACHINE`.

sub_key is a string that identifies the subkey to load.

file_name is the name of the file to load registry data from. This file must have been created with the `SaveKey()` function. Under the file allocation table (FAT) file system, the filename may not have an extension.

A call to `LoadKey()` fails if the calling process does not have the `SE_RESTORE_PRIVILEGE` privilege. Note that privileges are different from permissions – see the RegLoadKey documentation for more details.

If *key* is a handle returned by `ConnectRegistry()`, then the path specified in *file_name* is relative to the remote computer.

winreg.**OpenKey**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)
winreg.**OpenKeyEx**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)
Opens the specified key, returning a *handle object*.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that identifies the sub_key to open.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is *KEY_READ*. See *Access Rights* for other allowed values.

The result is a new handle to the specified key.

If the function fails, *OSError* is raised.

Alterado na versão 3.2: Allow the use of named arguments.

Alterado na versão 3.3: See *above*.

winreg.**QueryInfoKey**(*key*)
Returns information about a key, as a tuple.

key is an already open key, or one of the predefined *HKEY_* constants*.

The result is a tuple of 3 items:

Index	Significado
0	An integer giving the number of sub keys this key has.
1	An integer giving the number of values this key has.
2	An integer giving when the key was last modified (if available) as 100's of nanoseconds since Jan 1, 1601.

winreg.**QueryValue**(*key*, *sub_key*)
Retrieves the unnamed value for a key, as a string.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that holds the name of the subkey with which the value is associated. If this parameter is None or empty, the function retrieves the value set by the *SetValue()* method for the key identified by *key*.

Values in the registry have name, type, and data components. This method retrieves the data for a key's first value that has a NULL name. But the underlying API call doesn't return the type, so always use *QueryValueEx()* if possible.

winreg.**QueryValueEx**(*key*, *value_name*)
Retrieves the type and data for a specified value name associated with an open registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value_name is a string indicating the value to query.

The result is a tuple of 2 items:

Index	Significado
0	The value of the registry item.
1	An integer giving the registry type for this value (see table in docs for *SetValueEx()*)

winreg.**SaveKey**(*key*, *file_name*)
Saves the specified key, and all its subkeys to the specified file.

key is an already open key, or one of the predefined *HKEY_* constants*.

file_name is the name of the file to save registry data to. This file cannot already exist. If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the `LoadKey()` method.

If *key* represents a key on a remote computer, the path described by *file_name* is relative to the remote computer. The caller of this method must possess the `SeBackupPrivilege` security privilege. Note that privileges are different than permissions – see the Conflicts Between User Rights and Permissions documentation for more details.

This function passes `NULL` for *security_attributes* to the API.

winreg.**SetValue**(*key*, *sub_key*, *type*, *value*)
Associates a value with a specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

sub_key is a string that names the subkey with which the value is associated.

type is an integer that specifies the type of the data. Currently this must be `REG_SZ`, meaning only strings are supported. Use the `SetValueEx()` function for support for other data types.

value is a string that specifies the new value.

If the key specified by the *sub_key* parameter does not exist, the SetValue function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

The key identified by the *key* parameter must have been opened with `KEY_SET_VALUE` access.

winreg.**SetValueEx**(*key*, *value_name*, *reserved*, *type*, *value*)
Stores data in the value field of an open registry key.

key is an already open key, or one of the predefined *HKEY_* constants*.

value_name is a string that names the subkey with which the value is associated.

reserved can be anything – zero is always passed to the API.

type is an integer that specifies the type of the data. See *Value Types* for the available types.

value is a string that specifies the new value.

This method can also set additional value and type information for the specified key. The key identified by the key parameter must have been opened with `KEY_SET_VALUE` access.

To open the key, use the `CreateKey()` or `OpenKey()` methods.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

winreg.**DisableReflectionKey**(*key*)
Disables registry reflection for 32-bit processes running on a 64-bit operating system.

key is an already open key, or one of the predefined *HKEY_* constants*.

Will generally raise `NotImplementedError` if executed on a 32-bit operating system.

If the key is not on the reflection list, the function succeeds but has no effect. Disabling reflection for a key does not affect reflection of any subkeys.

winreg.**EnableReflectionKey**(*key*)
Restores registry reflection for the specified disabled key.

key is an already open key, or one of the predefined *HKEY_* constants*.

Will generally raise `NotImplementedError` if executed on a 32-bit operating system.

Restoring reflection for a key does not affect reflection of any subkeys.

`winreg.`**`QueryReflectionKey`**(*key*)
Determines the reflection state for the specified key.

key is an already open key, or one of the predefined *HKEY_* constants*.

Returns `True` if reflection is disabled.

Will generally raise `NotImplementedError` if executed on a 32-bit operating system.

35.3.2 Constantes

The following constants are defined for use in many `_winreg` functions.

HKEY_* Constants

`winreg.`**`HKEY_CLASSES_ROOT`**
Registry entries subordinate to this key define types (or classes) of documents and the properties associated with those types. Shell and COM applications use the information stored under this key.

`winreg.`**`HKEY_CURRENT_USER`**
Registry entries subordinate to this key define the preferences of the current user. These preferences include the settings of environment variables, data about program groups, colors, printers, network connections, and application preferences.

`winreg.`**`HKEY_LOCAL_MACHINE`**
Registry entries subordinate to this key define the physical state of the computer, including data about the bus type, system memory, and installed hardware and software.

`winreg.`**`HKEY_USERS`**
Registry entries subordinate to this key define the default user configuration for new users on the local computer and the user configuration for the current user.

`winreg.`**`HKEY_PERFORMANCE_DATA`**
Registry entries subordinate to this key allow you to access performance data. The data is not actually stored in the registry; the registry functions cause the system to collect the data from its source.

`winreg.`**`HKEY_CURRENT_CONFIG`**
Contains information about the current hardware profile of the local computer system.

`winreg.`**`HKEY_DYN_DATA`**
This key is not used in versions of Windows after 98.

Access Rights

For more information, see Registry Key Security and Access.

winreg.**KEY_ALL_ACCESS**
Combines the STANDARD_RIGHTS_REQUIRED, *KEY_QUERY_VALUE*, *KEY_SET_VALUE*, *KEY_CREATE_SUB_KEY*, *KEY_ENUMERATE_SUB_KEYS*, *KEY_NOTIFY*, and *KEY_CREATE_LINK* access rights.

winreg.**KEY_WRITE**
Combines the STANDARD_RIGHTS_WRITE, *KEY_SET_VALUE*, and *KEY_CREATE_SUB_KEY* access rights.

winreg.**KEY_READ**
Combines the STANDARD_RIGHTS_READ, *KEY_QUERY_VALUE*, *KEY_ENUMERATE_SUB_KEYS*, and *KEY_NOTIFY* values.

winreg.**KEY_EXECUTE**
Equivalent to *KEY_READ*.

winreg.**KEY_QUERY_VALUE**
Required to query the values of a registry key.

winreg.**KEY_SET_VALUE**
Required to create, delete, or set a registry value.

winreg.**KEY_CREATE_SUB_KEY**
Required to create a subkey of a registry key.

winreg.**KEY_ENUMERATE_SUB_KEYS**
Required to enumerate the subkeys of a registry key.

winreg.**KEY_NOTIFY**
Required to request change notifications for a registry key or for subkeys of a registry key.

winreg.**KEY_CREATE_LINK**
Reserved for system use.

64-bit Specific

For more information, see Accessing an Alternate Registry View.

winreg.**KEY_WOW64_64KEY**
Indicates that an application on 64-bit Windows should operate on the 64-bit registry view.

winreg.**KEY_WOW64_32KEY**
Indicates that an application on 64-bit Windows should operate on the 32-bit registry view.

Value Types

For more information, see Registry Value Types.

winreg.**REG_BINARY**
Binary data in any form.

winreg.**REG_DWORD**
Número de 32 bits.

winreg.**REG_DWORD_LITTLE_ENDIAN**
A 32-bit number in little-endian format. Equivalent to *REG_DWORD*.

winreg.**REG_DWORD_BIG_ENDIAN**
A 32-bit number in big-endian format.

winreg.**REG_EXPAND_SZ**
Null-terminated string containing references to environment variables (`%PATH%`).

winreg.**REG_LINK**
A Unicode symbolic link.

winreg.**REG_MULTI_SZ**
A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination automatically.)

winreg.**REG_NONE**
No defined value type.

winreg.**REG_QWORD**
A 64-bit number.

Novo na versão 3.6.

winreg.**REG_QWORD_LITTLE_ENDIAN**
A 64-bit number in little-endian format. Equivalent to *REG_QWORD*.

Novo na versão 3.6.

winreg.**REG_RESOURCE_LIST**
A device-driver resource list.

winreg.**REG_FULL_RESOURCE_DESCRIPTOR**
A hardware setting.

winreg.**REG_RESOURCE_REQUIREMENTS_LIST**
A hardware resource list.

winreg.**REG_SZ**
A null-terminated string.

35.3.3 Registry Handle Objects

This object wraps a Windows HKEY object, automatically closing it when the object is destroyed. To guarantee cleanup, you can call either the *Close()* method on the object, or the *CloseKey()* function.

All registry functions in this module return one of these objects.

All registry functions in this module which accept a handle object also accept an integer, however, use of the handle object is encouraged.

Handle objects provide semantics for `__bool__()` – thus

```
if handle:
    print("Yes")
```

will print `Yes` if the handle is currently valid (has not been closed or detached).

The object also support comparison semantics, so handle objects will compare true if they both reference the same underlying Windows handle value.

Handle objects can be converted to an integer (e.g., using the built-in *int()* function), in which case the underlying Windows handle value is returned. You can also use the *Detach()* method to return the integer handle, and also disconnect the Windows handle from the handle object.

PyHKEY.**Close**()
Closes the underlying Windows handle.

If the handle is already closed, no error is raised.

PyHKEY.**Detach**()
Detaches the Windows handle from the handle object.

The result is an integer that holds the value of the handle before it is detached. If the handle is already detached or closed, this will return zero.

After calling this function, the handle is effectively invalidated, but the handle is not closed. You would call this function when you need the underlying Win32 handle to exist beyond the lifetime of the handle object.

PyHKEY.**__enter__**()
PyHKEY.**__exit__**(**exc_info*)
The HKEY object implements __enter__() and __exit__() and thus supports the context protocol for the with statement:

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ...  # work with key
```

will automatically close *key* when control leaves the with block.

35.4 winsound — Interface de reprodução de som para Windows

O módulo *winsound* fornece acesso ao mecanismo básico de reprodução de som fornecido pelas plataformas Windows. Inclui funções e várias constantes.

winsound.**Beep**(*frequency*, *duration*)
Emite um bipe no alto-falante do PC. O parâmetro *frequency* especifica a frequência, em hertz, do som e deve estar no intervalo de 37 a 32.767. O parâmetro *duration* especifica o número de milissegundos que o som deve durar. Se o sistema não conseguir emitir um bipe no alto-falante, *RuntimeError* é levantado.

winsound.**PlaySound**(*sound*, *flags*)
Call the underlying PlaySound() function from the Platform API. The *sound* parameter may be a filename, a system sound alias, audio data as a *bytes-like object*, or None. Its interpretation depends on the value of *flags*, which can be a bitwise ORed combination of the constants described below. If the *sound* parameter is None, any currently playing waveform sound is stopped. If the system indicates an error, *RuntimeError* is raised.

winsound.**MessageBeep**(*type=MB_OK*)
Call the underlying MessageBeep() function from the Platform API. This plays a sound as specified in the registry. The *type* argument specifies which sound to play; possible values are -1, MB_ICONASTERISK, MB_ICONEXCLAMATION, MB_ICONHAND, MB_ICONQUESTION, and MB_OK, all described below. The value -1 produces a "simple beep"; this is the final fallback if a sound cannot be played otherwise. If the system indicates an error, *RuntimeError* is raised.

winsound.**SND_FILENAME**
The *sound* parameter is the name of a WAV file. Do not use with *SND_ALIAS*.

winsound.**SND_ALIAS**
The *sound* parameter is a sound association name from the registry. If the registry contains no such name, play the system default sound unless *SND_NODEFAULT* is also specified. If no default sound is registered, raise *RuntimeError*. Do not use with *SND_FILENAME*.

All Win32 systems support at least the following; most systems support many more:

PlaySound() name	Corresponding Control Panel Sound name
'SystemAsterisk'	Asterisk
'SystemExclamation'	Exclamation
'SystemExit'	Exit Windows
'SystemHand'	Critical Stop
'SystemQuestion'	Question

Por exemplo:

```
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (because
# "*" probably isn't the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

winsound.**SND_LOOP**
Play the sound repeatedly. The *SND_ASYNC* flag must also be used to avoid blocking. Cannot be used with *SND_MEMORY*.

winsound.**SND_MEMORY**
The *sound* parameter to *PlaySound()* is a memory image of a WAV file, as a *bytes-like object*.

Nota: This module does not support playing from a memory image asynchronously, so a combination of this flag and *SND_ASYNC* will raise *RuntimeError*.

winsound.**SND_PURGE**
Stop playing all instances of the specified sound.

Nota: This flag is not supported on modern Windows platforms.

winsound.**SND_ASYNC**
Return immediately, allowing sounds to play asynchronously.

winsound.**SND_NODEFAULT**
If the specified sound cannot be found, do not play the system default sound.

winsound.**SND_NOSTOP**
Do not interrupt sounds currently playing.

winsound.**SND_NOWAIT**
Return immediately if the sound driver is busy.

Nota: This flag is not supported on modern Windows platforms.

winsound.**MB_ICONASTERISK**
Play the SystemDefault sound.

winsound.**MB_ICONEXCLAMATION**
Play the SystemExclamation sound.

winsound.**MB_ICONHAND**
Play the SystemHand sound.

winsound.**MB_ICONQUESTION**
Play the SystemQuestion sound.

winsound.**MB_OK**
Play the SystemDefault sound.

CAPÍTULO 36

Serviços Específicos Unix

Os módulos descritos neste capítulo fornecem interfaces para recursos que são exclusivos do sistema operacional Unix ou, em alguns casos, para algumas ou várias variantes do mesmo. Aqui está uma visão geral:

36.1 `posix` — As chamadas de sistema mais comuns do POSIX

Este módulo fornece acesso à funcionalidade do sistema operacional padronizada pelo padrão C e pelo padrão POSIX (uma interface Unix levemente disfarçada).

Não importe este módulo diretamente. Em vez disso, importe o módulo *os*, que fornece uma versão *portátil* dessa interface. No Unix, o módulo *os* fornece um superconjunto da interface *posix*. Em sistemas operacionais não Unix, o módulo *posix* não está disponível, mas um subconjunto está sempre disponível na interface *os*. Uma vez que *os* é importado, seu uso *não* causa penalidade de desempenho em comparação com *posix*. Além disso, *os* fornece algumas funcionalidades adicionais, como chamar automaticamente *putenv()* quando uma entrada em `os.environ` é alterada.

Erros são relatados como exceções. As exceções usuais são dadas para erros de tipo, enquanto os erros relatados pelas chamadas do sistema aumentam *OSError*.

36.1.1 Suporte a arquivos grandes

Vários sistemas operacionais (incluindo AIX, HP-UX, Irix e Solaris) fornecem suporte a arquivos maiores que 2 GiB a partir de um modelo de programação C em que `int` e `long` são valores de 32 bits. Isso geralmente é realizado definindo o tamanho relevante e os tipos de deslocamento como valores de 64 bits. Esses arquivos às vezes são chamados de *arquivos grandes*.

O suporte a arquivos grandes é ativado no Python quando o tamanho de um `off_t` é maior que a `long` e `long long` é pelo menos tão grande quanto um **:c:tipo:‘off_t‘**. Pode ser necessário configurar e compilar o Python com certos sinalizadores do compilador para ativar esse modo. Por exemplo, ele é ativado por padrão nas versões recentes do Irix, mas com o Solaris 2.6 e 2.7 você precisa fazer algo como:

```
CFLAGS="`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \
        ./configure
```

Em sistemas Linux com capacidade para arquivos grandes, isso pode funcionar:

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \
        ./configure
```

36.1.2 Conteúdo notável do módulo

Além de muitas funções descritas na documentação do módulo *os*, *posix* define o seguinte item de dados:

posix.**environ**
: Um dicionário que representa o ambiente de strings no momento em que o interpretador foi iniciado. Chaves e valores são bytes no Unix e str no Windows. Por exemplo, `environ[b'HOME']` (`environ['HOME']` no Windows) é o nome do caminho do diretório inicial, equivalente a `getenv("HOME")` em C .

  A modificação deste dicionário não afeta o ambiente de strings passado por *execv()*, *popen()* ou *system()*. Se você precisar alterar o ambiente, passe `environ` para *execve()* ou adicione atribuições de variável e instruções de exportação para a string de comando para *system()* ou *popen()*.

  Alterado na versão 3.2: No Unix, chaves e valores são bytes.

  Nota: O módulo *os* fornece uma implementação alternativa de `environ` que atualiza o ambiente ao ocorrerem modificações. Observe também que a atualização de *os.environ* tornará este dicionário obsoleto. O uso do módulo *os* é recomendado sobre o acesso direto ao módulo *posix*.

36.2 pwd — A senha do banco de dados

Este módulo provê acesso ao banco de dados das contas de usuário do sistema e suas respectivas senhas. Isto está disponível para todas as versões do Unix.

As entradas do banco de dados de senhas são reportadas como um objeto do tipo tupla, cujos atributos correspondem aos membros da estrutura `passwd` (Campos dos atributos abaixo, veja `<pwd.h>`):

Index	Atributo	Significado
0	`pw_name`	Nome de acesso
1	`pw_passwd`	Senha encriptada opcional
2	`pw_uid`	ID numérico do usuário
3	`pw_gid`	ID numérico do grupo
4	`pw_gecos`	Nome do usuário ou campor de comentário
5	`pw_dir`	Diretório home do usuário
6	`pw_shell`	Interpretador de comandos do usuário

O uid e o gid são números inteiros, e os outros são strings. *KeyError* é lançada se o campo requerido não puder ser encontrado.

Nota: Em Unix tradicional, o campo `pw_passwd` geralmente contém uma senha encriptada com um algoritmo derivado de DES (veja o módulo *crypt*). No entanto, a maioria dos Unixes modernos usam o chamado sistema *shadow password*.

Nesses Unixes o campo *pw_passwd* só contém um asterisco (`'*'`) ou a letra `'x'` e a senha encriptada é guardada no arquivo `/etc/shadow` o qual não é permitido o acesso irrestrito a leitura. Se o campo *pw_passwd* contém alguma coisa útil dependerá do sistema. Se disponível, o módulo *spwd* deve ser usado para acessar onde a senha for requerida.

Isto define os seguintes itens

`pwd.`**`getpwuid`**`(`*uid*`)`
Retorna a entrada do banco de dados de senhas para um dado ID de usuário

`pwd.`**`getpwnam`**`(`*name*`)`
Retorna a entrada do banco de dados de senhas para um dado nome de usuário

`pwd.`**`getpwall`**`()`
Retorna uma lista de todos as entradas disponíveis no banco de dados de senhas, em uma ordem arbitrária.

Ver também:

Módulo *grp* Uma interface para o banco de dados do grupo, similar a essa.

Módulo *spwd* Uma interface para o banco de dados de shadow passwords, similar a essa.

36.3 `spwd` — O banco de dados de senhas shadow

Este módulo fornece acesso ao banco de dados de senhas shadow do Unix. Está disponível em várias versões do Unix.

Você deve ter privilégios suficientes para acessar o banco de dados de senhas shadow (isso geralmente significa que você precisa ser root).

As entradas do banco de dados de senhas shadow são relatadas como um objeto tupla ou similar, cujos atributos correspondem aos membros da estrutura `spwd` (campo Atributo abaixo, consulte `<shadow.h>`):

Index	Atributo	Significado
0	`sp_namp`	Nome de acesso
1	`sp_pwdp`	Senha criptografada
2	`sp_lstchg`	Data da última alteração
3	`sp_min`	Número mínimo de dias entre alterações
4	`sp_max`	Número máximo de dias entre alterações
5	`sp_warn`	Número de dias antes da senha expirar para avisar o usuário sobre ela
6	`sp_inact`	Número de dias após a senha expirar até a conta ser desativada
7	`sp_expire`	Número de dias desde 1970-01-01 em que a conta expira
8	`sp_flag`	Reservado

Os itens sp_namp e sp_pwdp são strings, todos os outros são números inteiros. *KeyError* é levantada se a entrada solicitada não puder ser encontrada.

As seguintes funções são definidas:

`spwd.`**`getspnam`**`(`*name*`)`
Retorna a entrada do banco de dados de senhas shadow para o nome de usuário especificado.

Alterado na versão 3.6: Levanta um *PermissionError* em vez de *KeyError* se o usuário não tiver privilégios.

`spwd.`**`getspall`**`()`
Retorna uma lista de todas as entradas disponíveis do banco de dados de senhas shadow, em ordem arbitrária.

Ver também:

Módulo ***grp*** Uma interface para o banco de dados do grupo, similar a essa.

Módulo ***pwd*** Uma interface para o banco de dados de senhas normais, similar a esta.

36.4 `grp` — The group database

This module provides access to the Unix group database. It is available on all Unix versions.

Group database entries are reported as a tuple-like object, whose attributes correspond to the members of the `group` structure (Attribute field below, see `<pwd.h>`):

Index	Atributo	Significado
0	gr_name	o nome do grupo
1	gr_passwd	the (encrypted) group password; often empty
2	gr_gid	O ID numérico do grupo
3	gr_mem	all the group member's user names

The gid is an integer, name and password are strings, and the member list is a list of strings. (Note that most users are not explicitly listed as members of the group they are in according to the password database. Check both databases to get complete membership information. Also note that a `gr_name` that starts with a + or − is likely to be a YP/NIS reference and may not be accessible via *`getgrnam()`* or *`getgrgid()`*.)

Isto define os seguintes itens

`grp.`**`getgrgid`**(*gid*)
Return the group database entry for the given numeric group ID. *`KeyError`* is raised if the entry asked for cannot be found.

Obsoleto desde a versão 3.6: Since Python 3.6 the support of non-integer arguments like floats or strings in *`getgrgid()`* is deprecated.

`grp.`**`getgrnam`**(*name*)
Return the group database entry for the given group name. *`KeyError`* is raised if the entry asked for cannot be found.

`grp.`**`getgrall`**()
Return a list of all available group entries, in arbitrary order.

Ver também:

Módulo ***pwd*** An interface to the user database, similar to this.

Módulo ***spwd*** Uma interface para o banco de dados de shadow passwords, similar a essa.

36.5 crypt — Function to check Unix passwords

Código Fonte: Lib/crypt.py

This module implements an interface to the *crypt(3)* routine, which is a one-way hash function based upon a modified DES algorithm; see the Unix man page for further details. Possible uses include storing hashed passwords so you can check passwords without storing the actual password, or attempting to crack Unix passwords with a dictionary.

Notice that the behavior of this module depends on the actual implementation of the *crypt(3)* routine in the running system. Therefore, any extensions available on the current implementation will also be available on this module.

36.5.1 Hashing Methods

Novo na versão 3.3.

The *crypt* module defines the list of hashing methods (not all methods are available on all platforms):

crypt.**METHOD_SHA512**
: A Modular Crypt Format method with 16 character salt and 86 character hash based on the SHA-512 hash function. This is the strongest method.

crypt.**METHOD_SHA256**
: Another Modular Crypt Format method with 16 character salt and 43 character hash based on the SHA-256 hash function.

crypt.**METHOD_BLOWFISH**
: Another Modular Crypt Format method with 22 character salt and 31 character hash based on the Blowfish cipher.

  Novo na versão 3.7.

crypt.**METHOD_MD5**
: Another Modular Crypt Format method with 8 character salt and 22 character hash based on the MD5 hash function.

crypt.**METHOD_CRYPT**
: The traditional method with a 2 character salt and 13 characters of hash. This is the weakest method.

36.5.2 Atributos do módulo

Novo na versão 3.3.

crypt.**methods**
: A list of available password hashing algorithms, as `crypt.METHOD_*` objects. This list is sorted from strongest to weakest.

36.5.3 Module Functions

The *crypt* module defines the following functions:

crypt.**crypt**(*word*, *salt=None*)
: *word* will usually be a user's password as typed at a prompt or in a graphical interface. The optional *salt* is either a string as returned from *mksalt()*, one of the `crypt.METHOD_*` values (though not all may be available on all platforms), or a full encrypted password including salt, as returned by this function. If *salt* is not provided, the strongest method will be used (as returned by *methods()*).

Checking a password is usually done by passing the plain-text password as *word* and the full results of a previous *crypt()* call, which should be the same as the results of this call.

salt (either a random 2 or 16 character string, possibly prefixed with `$digit$` to indicate the method) which will be used to perturb the encryption algorithm. The characters in *salt* must be in the set `[./a-zA-Z0-9]`, with the exception of Modular Crypt Format which prefixes a `$digit$`.

Returns the hashed password as a string, which will be composed of characters from the same alphabet as the salt.

Since a few *crypt(3)* extensions allow different values, with different sizes in the *salt*, it is recommended to use the full crypted password as salt when checking for a password.

Alterado na versão 3.3: Accept `crypt.METHOD_*` values in addition to strings for *salt*.

`crypt.`**`mksalt`**(*method=None*, *, *rounds=None*)

Return a randomly generated salt of the specified method. If no *method* is given, the strongest method available as returned by *methods()* is used.

The return value is a string suitable for passing as the *salt* argument to *crypt()*.

rounds specifies the number of rounds for `METHOD_SHA256`, `METHOD_SHA512` and `METHOD_BLOWFISH`. For `METHOD_SHA256` and `METHOD_SHA512` it must be an integer between `1000` and `999_999_999`, the default is `5000`. For `METHOD_BLOWFISH` it must be a power of two between `16` (2^4) and `2_147_483_648` (2^{31}), the default is `4096` (2^{12}).

Novo na versão 3.3.

Alterado na versão 3.7: Added the *rounds* parameter.

36.5.4 Exemplos

A simple example illustrating typical use (a constant-time comparison operation is needed to limit exposure to timing attacks. *hmac.compare_digest()* is suitable for this purpose):

```
import pwd
import crypt
import getpass
from hmac import compare_digest as compare_hash

def login():
    username = input('Python login: ')
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
        if cryptedpasswd == 'x' or cryptedpasswd == '*':
            raise ValueError('no support for shadow passwords')
        cleartext = getpass.getpass()
        return compare_hash(crypt.crypt(cleartext, cryptedpasswd), cryptedpasswd)
    else:
        return True
```

To generate a hash of a password using the strongest available method and check it against the original:

```
import crypt
from hmac import compare_digest as compare_hash

hashed = crypt.crypt(plaintext)
if not compare_hash(hashed, crypt.crypt(plaintext, hashed)):
    raise ValueError("hashed version doesn't validate against original")
```

36.6 termios — Controle de tty no estilo POSIX

Este módulo fornece uma interface para as chamadas POSIX para controle de E/S do tty. Para uma descrição completa dessas chamadas, consulte a página de manual Unix *termios(3)*. Está disponível apenas para as versões Unix que tenham suporte ao controle de E/S de tty no estilo POSIX do *termios* configurado durante a instalação.

Todas as funções neste módulo usam um descritor de arquivo *fd* como seu primeiro argumento. Pode ser um descritor de arquivo de tipo inteiro, como retornado por `sys.stdin.fileno()`, ou um *objeto arquivo*, como o próprio `sys.stdin`.

Este módulo também define todas as constantes necessárias para trabalhar com as funções fornecidas aqui; estes têm o mesmo nome de seus equivalentes em C. Consulte a documentação do sistema para mais informações sobre o uso dessas interfaces de controle de terminal.

O módulo define as seguintes funções:

`termios.`**`tcgetattr`**(*fd*)
: Retorna uma lista contendo os atributos tty para o descritor de arquivo *fd*, da seguinte forma: `[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]` onde *cc* é uma lista dos caracteres especiais do tty (cada uma string de comprimento 1, exceto os itens com índices `VMIN` e `VTIME`, que são números inteiros quando esses campos são definidos). A interpretação dos sinalizadores e as velocidades, bem como a indexação no vetor *cc*, devem ser feitas usando as constantes simbólicas definidas no módulo *termios*.

`termios.`**`tcsetattr`**(*fd*, *when*, *attributes*)
: Define os atributos tty para o descritor de arquivo *fd* a partir de *attribute*, que é uma lista como a retornada por *tcgetattr()*. O argumento *when* determina quando os atributos são alterados: `TCSANOW` para mudar imediatamente, `TCSADRAIN` para alterar após transmitir todas as saídas na fila ou `TCSAFLUSH` para alterar após transmitir todas as saídas na fila e descartando todas as entradas na fila.

`termios.`**`tcsendbreak`**(*fd*, *duration*)
: Envia uma quebra no descritor de arquivo *fd*. Uma duração zero, representada por *duration*, envia uma pausa por 0,25 a 0,5 segundos; *duration* com valor diferente de zero tem um significado dependente do sistema.

`termios.`**`tcdrain`**(*fd*)
: Aguarda até que toda a saída escrita no descritor de arquivo *fd* seja transmitida.

`termios.`**`tcflush`**(*fd*, *queue*)
: Descarta dados na fila no descritor de arquivo *fd*. O seletor *queue* especifica qual fila: `TCIFLUSH` para a fila de entrada, `TCOFLUSH` para a fila de saída ou `TCIOFLUSH` para as duas filas.

`termios.`**`tcflow`**(*fd*, *action*)
: Suspende ou retoma a entrada ou saída no descritor de arquivo *fd*. O argumento *action* pode ser `TCOOFF` para suspender a saída, `TCOON` para reiniciar a saída, `TCIOFF` para suspender a entrada ou `TCION` para reiniciar a entrada.

Ver também:

Módulo *tty* Funções de conveniência para operações comuns de controle de terminal.

36.6.1 Exemplo

Aqui está uma função que solicita uma senha com o eco desativado. Observe a técnica usando uma chamada separada *tcgetattr()* e uma instrução `try` … `finally` para garantir que os atributos tty antigos sejam restaurados exatamente, aconteça o que acontecer:

```
def getpass(prompt="Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

36.7 `tty` — Funções de controle de terminal

Código Fonte: Lib/tty.py

O módulo *tty* define funções para colocar o tty nos modos de cbreak e não tratados (raw).

Por requerer o módulo *termios*, ele funcionará apenas no Unix.

O módulo *tty* define as seguintes funções:

`tty.`**`setraw`**(*fd*, *when=termios.TCSAFLUSH*)
: Altera o modo do descritor de arquivo *fd* para não tratado (raw). Se *when* for omitido, o padrão é `termios.TCSAFLUSH` e é passado para *termios.tcsetattr()*.

`tty.`**`setcbreak`**(*fd*, *when=termios.TCSAFLUSH*)
: Altera o modo do descritor de arquivo *fd* para cbreak. Se *when* for omitido, o padrão é `termios.TCSAFLUSH` e é passado para *termios.tcsetattr()*.

Ver também:

Módulo *termios* Interface baixo nível para controle de terminal.

36.8 `pty` — Pseudo-terminal utilities

Código Fonte: Lib/pty.py

The *pty* module defines operations for handling the pseudo-terminal concept: starting another process and being able to write to and read from its controlling terminal programmatically.

Because pseudo-terminal handling is highly platform dependent, there is code to do it only for Linux. (The Linux code is supposed to work on other platforms, but hasn't been tested yet.)

The *pty* module defines the following functions:

pty.**fork**()
Fork. Connect the child's controlling terminal to a pseudo-terminal. Return value is `(pid, fd)`. Note that the child gets *pid* 0, and the *fd* is *invalid*. The parent's return value is the *pid* of the child, and *fd* is a file descriptor connected to the child's controlling terminal (and also to the child's standard input and output).

pty.**openpty**()
Open a new pseudo-terminal pair, using *os.openpty()* if possible, or emulation code for generic Unix systems. Return a pair of file descriptors `(master, slave)`, for the master and the slave end, respectively.

pty.**spawn**(*argv*[, *master_read*[, *stdin_read*]])
Spawn a process, and connect its controlling terminal with the current process's standard io. This is often used to baffle programs which insist on reading from the controlling terminal. It is expected that the process spawned behind the pty will eventually terminate, and when it does *spawn* will return.

The functions *master_read* and *stdin_read* are passed a file descriptor which they should read from, and they should always return a byte string. In order to force spawn to return before the child process exits an *OSError* should be thrown.

The default implementation for both functions will read and return up to 1024 bytes each time the function is called. The *master_read* callback is passed the pseudoterminal's master file descriptor to read output from the child process, and *stdin_read* is passed file descriptor 0, to read from the parent process's standard input.

Returning an empty byte string from either callback is interpreted as an end-of-file (EOF) condition, and that callback will not be called after that. If *stdin_read* signals EOF the controlling terminal can no longer communicate with the parent process OR the child process. Unless the child process will quit without any input, *spawn* will then loop forever. If *master_read* signals EOF the same behavior results (on linux at least).

If both callbacks signal EOF then *spawn* will probably never return, unless *select* throws an error on your platform when passed three empty lists. This is a bug, documented in issue 26228.

Alterado na versão 3.4: *spawn()* now returns the status value from *os.waitpid()* on the child process.

36.8.1 Exemplo

The following program acts like the Unix command *script(1)*, using a pseudo-terminal to record all input and output of a terminal session in a "typescript".

```
import argparse
import os
import pty
import sys
import time

parser = argparse.ArgumentParser()
parser.add_argument('-a', dest='append', action='store_true')
parser.add_argument('-p', dest='use_python', action='store_true')
parser.add_argument('filename', nargs='?', default='typescript')
options = parser.parse_args()

shell = sys.executable if options.use_python else os.environ.get('SHELL', 'sh')
filename = options.filename
mode = 'ab' if options.append else 'wb'

with open(filename, mode) as script:
    def read(fd):
        data = os.read(fd, 1024)
        script.write(data)
        return data
```

(continua na próxima página)

(continuação da página anterior)

```
    print('Script started, file is', filename)
    script.write(('Script started on %s\n' % time.asctime()).encode())

    pty.spawn(shell, read)

    script.write(('Script done on %s\n' % time.asctime()).encode())
    print('Script done, file is', filename)
```

36.9 `fcntl` — as chamadas do sistema `fcntl` e `ioctl`

Este módulo executa o controle de arquivos e o controle de I/O em descritores de arquivos. É uma interface para as rotinas `fcntl()` and `ioctl()` do Unix. Para obter uma descrição completa dessas chamadas, consulte *`fcntl(2)`* e *`ioctl(2)`* Páginas do Manual do Unix.

All functions in this module take a file descriptor *fd* as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or an *`io.IOBase`* object, such as `sys.stdin` itself, which provides a *`fileno()`* that returns a genuine file descriptor.

Alterado na versão 3.3: Operations in this module used to raise an *`IOError`* where they now raise an *`OSError`*.

O módulo define as seguintes funções:

`fcntl.`**`fcntl`**(*fd*, *cmd*, *arg=0*)

Perform the operation *cmd* on file descriptor *fd* (file objects providing a *`fileno()`* method are accepted as well). The values used for *cmd* are operating system dependent, and are available as constants in the *`fcntl`* module, using the same names as used in the relevant C header files. The argument *arg* can either be an integer value, or a *`bytes`* object. With an integer value, the return value of this function is the integer return value of the C `fcntl()` call. When the argument is bytes it represents a binary structure, e.g. created by *`struct.pack()`*. The binary data is copied to a buffer whose address is passed to the C `fcntl()` call. The return value after a successful call is the contents of the buffer, converted to a *`bytes`* object. The length of the returned object will be the same as the length of the *arg* argument. This is limited to 1024 bytes. If the information returned in the buffer by the operating system is larger than 1024 bytes, this is most likely to result in a segmentation violation or a more subtle data corruption.

Se o `fcntl()` vier a falhar, um exceção *`OSError`* é levantada.

`fcntl.`**`ioctl`**(*fd*, *request*, *arg=0*, *mutate_flag=True*)

This function is identical to the *`fcntl()`* function, except that the argument handling is even more complicated.

The *request* parameter is limited to values that can fit in 32-bits. Additional constants of interest for use as the *request* argument can be found in the *`termios`* module, under the same names as used in the relevant C header files.

The parameter *arg* can be one of an integer, an object supporting the read-only buffer interface (like *`bytes`*) or an object supporting the read-write buffer interface (like *`bytearray`*).

In all but the last case, behaviour is as for the *`fcntl()`* function.

If a mutable buffer is passed, then the behaviour is determined by the value of the *mutate_flag* parameter.

If it is false, the buffer's mutability is ignored and behaviour is as for a read-only buffer, except that the 1024 byte limit mentioned above is avoided – so long as the buffer you pass is at least as long as what the operating system wants to put there, things should work.

If *mutate_flag* is true (the default), then the buffer is (in effect) passed to the underlying *ioctl()* system call, the latter's return code is passed back to the calling Python, and the buffer's new contents reflect the action of the *ioctl()*. This is a slight simplification, because if the supplied buffer is less than 1024 bytes long it is first copied into a static buffer 1024 bytes long which is then passed to *ioctl()* and copied back into the supplied buffer.

If the `ioctl()` fails, an *OSError* exception is raised.

Um exemplo:

```
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, "  "))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
```

`fcntl.`**`flock`**(*fd*, *operation*)

Perform the lock operation *operation* on file descriptor *fd* (file objects providing a *fileno()* method are accepted as well). See the Unix manual *flock(2)* for details. (On some systems, this function is emulated using `fcntl()`.)

If the `flock()` fails, an *OSError* exception is raised.

`fcntl.`**`lockf`**(*fd*, *cmd*, *len=0*, *start=0*, *whence=0*)

This is essentially a wrapper around the *fcntl()* locking calls. *fd* is the file descriptor of the file to lock or unlock, and *cmd* is one of the following values:

- `LOCK_UN` – desbloquear
- `LOCK_SH` – acquire a shared lock
- `LOCK_EX` – acquire an exclusive lock

When *cmd* is `LOCK_SH` or `LOCK_EX`, it can also be bitwise ORed with `LOCK_NB` to avoid blocking on lock acquisition. If `LOCK_NB` is used and the lock cannot be acquired, an *OSError* will be raised and the exception will have an *errno* attribute set to `EACCES` or `EAGAIN` (depending on the operating system; for portability, check for both values). On at least some systems, `LOCK_EX` can only be used if the file descriptor refers to a file opened for writing.

len is the number of bytes to lock, *start* is the byte offset at which the lock starts, relative to *whence*, and *whence* is as with *io.IOBase.seek()*, specifically:

- `0` – relative to the start of the file (*os.SEEK_SET*)
- `1` – relative to the current buffer position (*os.SEEK_CUR*)
- `2` – relative to the end of the file (*os.SEEK_END*)

The default for *start* is 0, which means to start at the beginning of the file. The default for *len* is 0 which means to lock to the end of the file. The default for *whence* is also 0.

Examples (all on a SVR4 compliant system):

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)
```

(continua na próxima página)

(continuação da página anterior)

```
lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

Note that in the first example the return value variable *rv* will hold an integer value; in the second example it will hold a `bytes` object. The structure lay-out for the *lockdata* variable is system dependent — therefore using the `flock()` call may be better.

Ver também:

Módulo `os` If the locking flags `O_SHLOCK` and `O_EXLOCK` are present in the `os` module (on BSD only), the `os.open()` function provides an alternative to the `lockf()` and `flock()` functions.

36.10 `pipes` — Interface to shell pipelines

Código Fonte: Lib/pipes.py

The `pipes` module defines a class to abstract the concept of a *pipeline* — a sequence of converters from one file to another.

Because the module uses **/bin/sh** command lines, a POSIX or compatible shell for `os.system()` and `os.popen()` is required.

The `pipes` module defines the following class:

class pipes.**Template**
Uma abstração de um pipeline

Exemplo:

```
>>> import pipes
>>> t = pipes.Template()
>>> t.append('tr a-z A-Z', '--')
>>> f = t.open('pipefile', 'w')
>>> f.write('hello world')
>>> f.close()
>>> open('pipefile').read()
'HELLO WORLD'
```

36.10.1 Objetos Template

Template objects following methods:

Template.**reset**()
Restore a pipeline template to its initial state.

Template.**clone**()
Return a new, equivalent, pipeline template.

Template.**debug**(*flag*)
If *flag* is true, turn debugging on. Otherwise, turn debugging off. When debugging is on, commands to be executed are printed, and the shell is given `set -x` command to be more verbose.

Template.**append**(*cmd*, *kind*)
Append a new action at the end. The *cmd* variable must be a valid bourne shell command. The *kind* variable consists of two letters.

The first letter can be either of `'-'` (which means the command reads its standard input), `'f'` (which means the commands reads a given file on the command line) or `'.'` (which means the commands reads no input, and hence must be first.)

Similarly, the second letter can be either of `'-'` (which means the command writes to standard output), `'f'` (which means the command writes a file on the command line) or `'.'` (which means the command does not write anything, and hence must be last.)

Template.**prepend**(*cmd*, *kind*)
Add a new action at the beginning. See *append()* for explanations of the arguments.

Template.**open**(*file*, *mode*)
Return a file-like object, open to *file*, but read from or written to by the pipeline. Note that only one of `'r'`, `'w'` may be given.

Template.**copy**(*infile*, *outfile*)
Copy *infile* to *outfile* through the pipe.

36.11 resource — Resource usage information

This module provides basic mechanisms for measuring and controlling system resources utilized by a program.

Symbolic constants are used to specify particular system resources and to request usage information about either the current process or its children.

An *OSError* is raised on syscall failure.

exception resource.**error**
A deprecated alias of *OSError*.

Alterado na versão 3.3: Following **PEP 3151**, this class was made an alias of *OSError*.

36.11.1 Resource Limits

Resources usage can be limited using the *setrlimit()* function described below. Each resource is controlled by a pair of limits: a soft limit and a hard limit. The soft limit is the current limit, and may be lowered or raised by a process over time. The soft limit can never exceed the hard limit. The hard limit can be lowered to any value greater than the soft limit, but not raised. (Only processes with the effective UID of the super-user can raise a hard limit.)

The specific resources that can be limited are system dependent. They are described in the *getrlimit(2)* man page. The resources listed below are supported when the underlying operating system supports them; resources which cannot be checked or controlled by the operating system are not defined in this module for those platforms.

resource.**RLIM_INFINITY**
Constant used to represent the limit for an unlimited resource.

resource.**getrlimit**(*resource*)
Returns a tuple `(soft, hard)` with the current soft and hard limits of *resource*. Raises *ValueError* if an invalid resource is specified, or *error* if the underlying system call fails unexpectedly.

resource.**setrlimit**(*resource*, *limits*)
Sets new limits of consumption of *resource*. The *limits* argument must be a tuple `(soft, hard)` of two integers describing the new limits. A value of *RLIM_INFINITY* can be used to request a limit that is unlimited.

Raises *ValueError* if an invalid resource is specified, if the new soft limit exceeds the hard limit, or if a process tries to raise its hard limit. Specifying a limit of *RLIM_INFINITY* when the hard or system limit for that resource is not unlimited will result in a *ValueError*. A process with the effective UID of super-user can request any valid limit value, including unlimited, but *ValueError* will still be raised if the requested limit exceeds the system imposed limit.

`setrlimit` may also raise *error* if the underlying system call fails.

resource.**prlimit**(*pid*, *resource*[, *limits*])
Combines *setrlimit()* and *getrlimit()* in one function and supports to get and set the resources limits of an arbitrary process. If *pid* is 0, then the call applies to the current process. *resource* and *limits* have the same meaning as in *setrlimit()*, except that *limits* is optional.

When *limits* is not given the function returns the *resource* limit of the process *pid*. When *limits* is given the *resource* limit of the process is set and the former resource limit is returned.

Raises *ProcessLookupError* when *pid* can't be found and *PermissionError* when the user doesn't have `CAP_SYS_RESOURCE` for the process.

Availability: Linux 2.6.36 or later with glibc 2.13 or later.

Novo na versão 3.4.

These symbols define resources whose consumption can be controlled using the *setrlimit()* and *getrlimit()* functions described below. The values of these symbols are exactly the constants used by C programs.

The Unix man page for *getrlimit(2)* lists the available resources. Note that not all systems use the same symbol or same value to denote the same resource. This module does not attempt to mask platform differences — symbols not defined for a platform will not be available from this module on that platform.

resource.**RLIMIT_CORE**
The maximum size (in bytes) of a core file that the current process can create. This may result in the creation of a partial core file if a larger core would be required to contain the entire process image.

resource.**RLIMIT_CPU**
The maximum amount of processor time (in seconds) that a process can use. If this limit is exceeded, a `SIGXCPU` signal is sent to the process. (See the *signal* module documentation for information about how to catch this signal and do something useful, e.g. flush open files to disk.)

resource.**RLIMIT_FSIZE**
The maximum size of a file which the process may create.

resource.**RLIMIT_DATA**
The maximum size (in bytes) of the process's heap.

resource.**RLIMIT_STACK**
The maximum size (in bytes) of the call stack for the current process. This only affects the stack of the main thread in a multi-threaded process.

resource.**RLIMIT_RSS**
The maximum resident set size that should be made available to the process.

resource.**RLIMIT_NPROC**
The maximum number of processes the current process may create.

resource.**RLIMIT_NOFILE**
The maximum number of open file descriptors for the current process.

resource.**RLIMIT_OFILE**
The BSD name for *RLIMIT_NOFILE*.

resource.**RLIMIT_MEMLOCK**
The maximum address space which may be locked in memory.

resource.**RLIMIT_VMEM**
The largest area of mapped memory which the process may occupy.

resource.**RLIMIT_AS**
The maximum area (in bytes) of address space which may be taken by the process.

resource.**RLIMIT_MSGQUEUE**
The number of bytes that can be allocated for POSIX message queues.

Availability: Linux 2.6.8 or later.

Novo na versão 3.4.

resource.**RLIMIT_NICE**
The ceiling for the process's nice level (calculated as 20 - rlim_cur).

Availability: Linux 2.6.12 or later.

Novo na versão 3.4.

resource.**RLIMIT_RTPRIO**
The ceiling of the real-time priority.

Availability: Linux 2.6.12 or later.

Novo na versão 3.4.

resource.**RLIMIT_RTTIME**
The time limit (in microseconds) on CPU time that a process can spend under real-time scheduling without making a blocking syscall.

Availability: Linux 2.6.25 or later.

Novo na versão 3.4.

resource.**RLIMIT_SIGPENDING**
The number of signals which the process may queue.

Availability: Linux 2.6.8 or later.

Novo na versão 3.4.

resource.**RLIMIT_SBSIZE**
The maximum size (in bytes) of socket buffer usage for this user. This limits the amount of network memory, and hence the amount of mbufs, that this user may hold at any time.

Availability: FreeBSD 9 or later.

Novo na versão 3.4.

resource.**RLIMIT_SWAP**
The maximum size (in bytes) of the swap space that may be reserved or used by all of this user id's processes. This limit is enforced only if bit 1 of the vm.overcommit sysctl is set. Please see *tuning(7)* for a complete description of this sysctl.

Availability: FreeBSD 9 or later.

Novo na versão 3.4.

resource.**RLIMIT_NPTS**
The maximum number of pseudo-terminals created by this user id.

Availability: FreeBSD 9 or later.

Novo na versão 3.4.

36.11.2 Resource Usage

These functions are used to retrieve resource usage information:

resource.**getrusage**(*who*)
This function returns an object that describes the resources consumed by either the current process or its children, as specified by the *who* parameter. The *who* parameter should be specified using one of the RUSAGE_* constants described below.

The fields of the return value each describe how a particular system resource has been used, e.g. amount of time spent running is user mode or number of times the process was swapped out of main memory. Some values are dependent on the clock tick internal, e.g. the amount of memory the process is using.

For backward compatibility, the return value is also accessible as a tuple of 16 elements.

The fields ru_utime and ru_stime of the return value are floating point values representing the amount of time spent executing in user mode and the amount of time spent executing in system mode, respectively. The remaining values are integers. Consult the *getrusage(2)* man page for detailed information about these values. A brief summary is presented here:

Index	Campo	Resource
0	ru_utime	time in user mode (float)
1	ru_stime	time in system mode (float)
2	ru_maxrss	maximum resident set size
3	ru_ixrss	shared memory size
4	ru_idrss	unshared memory size
5	ru_isrss	unshared stack size
6	ru_minflt	page faults not requiring I/O
7	ru_majflt	page faults requiring I/O
8	ru_nswap	number of swap outs
9	ru_inblock	block input operations
10	ru_oublock	block output operations
11	ru_msgsnd	messages sent
12	ru_msgrcv	messages received
13	ru_nsignals	signals received
14	ru_nvcsw	voluntary context switches
15	ru_nivcsw	involuntary context switches

This function will raise a *ValueError* if an invalid *who* parameter is specified. It may also raise *error* exception in unusual circumstances.

resource.**getpagesize**()
Returns the number of bytes in a system page. (This need not be the same as the hardware page size.)

The following RUSAGE_* symbols are passed to the *getrusage()* function to specify which processes information should be provided for.

resource.**RUSAGE_SELF**
Pass to *getrusage()* to request resources consumed by the calling process, which is the sum of resources used by all threads in the process.

resource.**RUSAGE_CHILDREN**
Pass to *getrusage()* to request resources consumed by child processes of the calling process which have been terminated and waited for.

resource.**RUSAGE_BOTH**
Pass to *getrusage()* to request resources consumed by both the current process and child processes. May not be available on all systems.

resource.**RUSAGE_THREAD**
Pass to *getrusage()* to request resources consumed by the current thread. May not be available on all systems.

Novo na versão 3.2.

36.12 nis — Interface para NIS da Sun (Yellow Pages)

O módulo *nis* fornece um invólucro fino em torno da biblioteca NIS, útil para administração central de vários hosts.

Como o NIS existe apenas em sistemas Unix, este módulo está disponível apenas para Unix.

O módulo *nis* define as seguintes funções:

nis.**match**(*key*, *mapname*, *domain=default_domain*)
Retorna a correspondência para *key* no mapa *mapname* ou levanta um erro (*nis.error*) se não houver nenhum. Ambos devem ser strings, *key* está limpo em 8 bits. O valor de retorno é uma matriz arbitrária de bytes (pode conter NULL e outras alegrias).

Observe que *mapname* é verificado primeiro se for um alias para outro nome.

O argumento *domain* permite substituir o domínio NIS usado para a pesquisa. Se não especificado, a pesquisa está no domínio NIS padrão.

nis.**cat**(*mapname*, *domain=default_domain*)
Retorna um mapeamento de dicionário de *key* para *value* de modo que `match(key, mapname)==value`. Observe que as chaves e os valores do dicionário são matrizes arbitrárias de bytes.

Observe que *mapname* é verificado primeiro se for um alias para outro nome.

O argumento *domain* permite substituir o domínio NIS usado para a pesquisa. Se não especificado, a pesquisa está no domínio NIS padrão.

nis.**maps**(*domain=default_domain*)
Retorna uma lista de todos os mapas válidos.

O argumento *domain* permite substituir o domínio NIS usado para a pesquisa. Se não especificado, a pesquisa está no domínio NIS padrão.

nis.**get_default_domain**()
Retorna o domínio NIS padrão do sistema.

O módulo *nis* define a exceção padrão:

exception nis.**error**
Um erro levantado quando uma função NIS retorna um código de erro.

36.13 `syslog` — Rotinas da biblioteca syslog do Unix

Este módulo fornece uma interface para as rotinas da biblioteca `syslog` do Unix. Consulte as páginas de manual do Unix para uma descrição detalhada do recurso `syslog`.

Este módulo é uma camada para a família de rotinas `syslog` do sistema. Uma biblioteca Python pura que pode se comunicar com um servidor syslog está disponível no módulo *`logging.handlers`* como `SysLogHandler`.

O módulo define as seguintes funções:

`syslog.`**`syslog`**(*message*)
`syslog.`**`syslog`**(*priority*, *message*)
Envia a string *message* para o registrador do sistema. Uma nova linha final é adicionada, se necessário. Cada mensagem é marcada com uma prioridade composta por um *facility* e um *level*. O argumento opcional *priority*, cujo padrão é `LOG_INFO`, determina a prioridade da mensagem. Se a facilidade não está codificada em *priority* usando o OU lógico (`LOG_INFO | LOG_USER`), o valor dado na chamada *`openlog()`* é usado.

Se *`openlog()`* não foi chamado antes da chamada para *`syslog()`*, `openlog()` será chamado sem argumentos.

`syslog.`**`openlog`**([*ident*[, *logoption*[, *facility*]]])
As opções de log das chamadas subsequentes *`syslog()`* podem ser definidas chamando *`openlog()`*. *`syslog()`* irá chamar *`openlog()`* sem argumentos se o log não estiver aberto no momento.

O argumento nomeado opcional *ident* é uma string que é prefixada a cada mensagem, e o padrão é `sys.argv[0]` com os componentes do caminho inicial removidos. O argumento nomeado opcional *logoption* (o padrão é 0) é um campo de bits – veja abaixo os valores possíveis para combinar. O argumento nomeado opcional *facility* (o padrão é `LOG_USER`) define o recurso padrão para mensagens que não possuem um recurso explicitamente codificado.

Alterado na versão 3.2: Nas versões anteriores, os argumentos nomeados não eram permitidos e *ident* era obrigatório. O padrão para *ident* era dependente das bibliotecas do sistema e, frequentemente, era `python` ao invés do nome do arquivo de programa Python.

`syslog.`**`closelog`**()
Redefine os valores do módulo syslog e chama a biblioteca de sistema `closelog()`.

Isso faz com que o módulo se comporte como quando importado inicialmente. Por exemplo, *`openlog()`* será chamado na primeira chamada *`syslog()`* (se *`openlog()`* ainda não foi chamado), e *ident* e outro *`openlog()`* os parâmetros são redefinidos para os padrões.

`syslog.`**`setlogmask`**(*maskpri*)
Define a máscara de prioridade como *maskpri* e retorna o valor da máscara anterior. Chamadas para *`syslog()`* com um nível de prioridade não definido em *maskpri* são ignoradas. O padrão é registrar todas as prioridades. A função `LOG_MASK(pri)` calcula a máscara para a prioridade individual *pri*. A função `LOG_UPTO(pri)` calcula a máscara para todas as prioridades até e incluindo *pri*.

O módulo define as seguintes constantes:

Níveis de prioridade (alto a baixo): `LOG_EMERG`, `LOG_ALERT`, `LOG_CRIT`, `LOG_ERR`, `LOG_WARNING`, `LOG_NOTICE`, `LOG_INFO`, `LOG_DEBUG`.

Facilidades: `LOG_KERN`, `LOG_USER`, `LOG_MAIL`, `LOG_DAEMON`, `LOG_AUTH`, `LOG_LPR`, `LOG_NEWS`, `LOG_UUCP`, `LOG_CRON`, `LOG_SYSLOG`, `LOG_LOCAL0` até `LOG_LOCAL7` e, se definido em `<syslog.h>`, `LOG_AUTHPRIV`.

Opções de log: `LOG_PID`, `LOG_CONS`, `LOG_NDELAY` e, se definido em `<syslog.h>`, `LOG_ODELAY`, `LOG_NOWAIT` e `LOG_PERROR`.

36.13.1 Exemplos

Exemplo simples

Um conjunto de exemplos simples:

```
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

Um exemplo de configuração de algumas opções de log, isso incluiria o ID do processo nas mensagens registradas e escreveria as mensagens no recurso de destino usado para o log de correio:

```
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```

CAPÍTULO 37

Módulos Substituídos

Os módulos descritos neste capítulo são obsoletos e mantidos apenas para compatibilidade com versões anteriores. Eles foram substituídos por outros módulos.

37.1 `optparse` — Parser for command line options

Código Fonte: Lib/optparse.py

Obsoleto desde a versão 3.2: The *optparse* module is deprecated and will not be developed further; development will continue with the *argparse* module.

---

optparse is a more convenient, flexible, and powerful library for parsing command-line options than the old *getopt* module. *optparse* uses a more declarative style of command-line parsing: you create an instance of *OptionParser*, populate it with options, and parse the command line. *optparse* allows users to specify options in the conventional GNU/POSIX syntax, and additionally generates usage and help messages for you.

Here's an example of using *optparse* in a simple script:

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

With these few lines of code, users of your script can now do the "usual thing" on the command-line, for example:

```
<yourscript> --file=outfile -q
```

As it parses the command line, *optparse* sets attributes of the `options` object returned by `parse_args()` based on user-supplied command-line values. When `parse_args()` returns from parsing this command line, `options.filename` will be `"outfile"` and `options.verbose` will be `False`. *optparse* supports both long and short options, allows short options to be merged together, and allows options to be associated with their arguments in a variety of ways. Thus, the following command lines are all equivalent to the above example:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

Additionally, users can run one of

```
<yourscript> -h
<yourscript> --help
```

and *optparse* will print out a brief summary of your script's options:

```
Usage: <yourscript> [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

where the value of *yourscript* is determined at runtime (normally from `sys.argv[0]`).

37.1.1 Background

optparse was explicitly designed to encourage the creation of programs with straightforward, conventional command-line interfaces. To that end, it supports only the most common command-line syntax and semantics conventionally used under Unix. If you are unfamiliar with these conventions, read this section to acquaint yourself with them.

Terminology

argumento a string entered on the command-line, and passed by the shell to `execl()` or `execv()`. In Python, arguments are elements of `sys.argv[1:]` (`sys.argv[0]` is the name of the program being executed). Unix shells also use the term "word".

It is occasionally desirable to substitute an argument list other than `sys.argv[1:]`, so you should read "argument" as "an element of `sys.argv[1:]`, or of some other list provided as a substitute for `sys.argv[1:]`".

option an argument used to supply extra information to guide or customize the execution of a program. There are many different syntaxes for options; the traditional Unix syntax is a hyphen ("-") followed by a single letter, e.g. `-x` or `-F`. Also, traditional Unix syntax allows multiple options to be merged into a single argument, e.g. `-x -F` is equivalent to `-xF`. The GNU project introduced `--` followed by a series of hyphen-separated words, e.g. `--file` or `--dry-run`. These are the only two option syntaxes provided by *optparse*.

Some other option syntaxes that the world has seen include:

- a hyphen followed by a few letters, e.g. `-pf` (this is *not* the same as multiple options merged into a single argument)

- a hyphen followed by a whole word, e.g. `-file` (this is technically equivalent to the previous syntax, but they aren't usually seen in the same program)
- a plus sign followed by a single letter, or a few letters, or a word, e.g. `+f`, `+rgb`
- a slash followed by a letter, or a few letters, or a word, e.g. `/f`, `/file`

These option syntaxes are not supported by *optparse*, and they never will be. This is deliberate: the first three are non-standard on any environment, and the last only makes sense if you're exclusively targeting VMS, MS-DOS, and/or Windows.

option argument an argument that follows an option, is closely associated with that option, and is consumed from the argument list when that option is. With *optparse*, option arguments may either be in a separate argument from their option:

```
-f foo
--file foo
```

or included in the same argument:

```
-ffoo
--file=foo
```

Typically, a given option either takes an argument or it doesn't. Lots of people want an "optional option arguments" feature, meaning that some options will take an argument if they see it, and won't if they don't. This is somewhat controversial, because it makes parsing ambiguous: if `-a` takes an optional argument and `-b` is another option entirely, how do we interpret `-ab`? Because of this ambiguity, *optparse* does not support this feature.

positional argument (argumento posicional) something leftover in the argument list after options have been parsed, i.e. after options and their arguments have been parsed and removed from the argument list.

required option an option that must be supplied on the command-line; note that the phrase "required option" is self-contradictory in English. *optparse* doesn't prevent you from implementing required options, but doesn't give you much help at it either.

For example, consider this hypothetical command-line:

```
prog -v --report report.txt foo bar
```

`-v` and `--report` are both options. Assuming that `--report` takes one argument, `report.txt` is an option argument. `foo` and `bar` are positional arguments.

What are options for?

Options are used to provide extra information to tune or customize the execution of a program. In case it wasn't clear, options are usually *optional*. A program should be able to run just fine with no options whatsoever. (Pick a random program from the Unix or GNU toolsets. Can it run without any options at all and still make sense? The main exceptions are `find`, `tar`, and `dd`—all of which are mutant oddballs that have been rightly criticized for their non-standard syntax and confusing interfaces.)

Lots of people want their programs to have "required options". Think about it. If it's required, then it's *not optional*! If there is a piece of information that your program absolutely requires in order to run successfully, that's what positional arguments are for.

As an example of good command-line interface design, consider the humble `cp` utility, for copying files. It doesn't make much sense to try to copy files without supplying a destination and at least one source. Hence, `cp` fails if you run it with no arguments. However, it has a flexible, useful syntax that does not require any options at all:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

You can get pretty far with just that. Most `cp` implementations provide a bunch of options to tweak exactly how the files are copied: you can preserve mode and modification time, avoid following symlinks, ask before clobbering existing files, etc. But none of this distracts from the core mission of `cp`, which is to copy either one file to another, or several files to another directory.

What are positional arguments for?

Positional arguments are for those pieces of information that your program absolutely, positively requires to run.

A good user interface should have as few absolute requirements as possible. If your program requires 17 distinct pieces of information in order to run successfully, it doesn't much matter *how* you get that information from the user—most people will give up and walk away before they successfully run the program. This applies whether the user interface is a command-line, a configuration file, or a GUI: if you make that many demands on your users, most of them will simply give up.

In short, try to minimize the amount of information that users are absolutely required to supply—use sensible defaults whenever possible. Of course, you also want to make your programs reasonably flexible. That's what options are for. Again, it doesn't matter if they are entries in a config file, widgets in the "Preferences" dialog of a GUI, or command-line options—the more options you implement, the more flexible your program is, and the more complicated its implementation becomes. Too much flexibility has drawbacks as well, of course; too many options can overwhelm users and make your code much harder to maintain.

37.1.2 Tutorial

While *optparse* is quite flexible and powerful, it's also straightforward to use in most cases. This section covers the code patterns that are common to any *optparse*-based program.

First, you need to import the OptionParser class; then, early in the main program, create an OptionParser instance:

```
from optparse import OptionParser
...
parser = OptionParser()
```

Then you can start defining options. The basic syntax is:

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

Each option has one or more option strings, such as `-f` or `--file`, and several option attributes that tell *optparse* what to expect and what to do when it encounters that option on the command line.

Typically, each option will have one short option string and one long option string, e.g.:

```
parser.add_option("-f", "--file", ...)
```

You're free to define as many short option strings and as many long option strings as you like (including zero), as long as there is at least one option string overall.

The option strings passed to *OptionParser.add_option()* are effectively labels for the option defined by that call. For brevity, we will frequently refer to *encountering an option* on the command line; in reality, *optparse* encounters *option strings* and looks up options from them.

Once all of your options are defined, instruct *optparse* to parse your program's command line:

```
(options, args) = parser.parse_args()
```

(If you like, you can pass a custom argument list to `parse_args()`, but that's rarely necessary: by default it uses `sys.argv[1:]`.)

`parse_args()` returns two values:

- `options`, an object containing values for all of your options—e.g. if `--file` takes a single string argument, then `options.file` will be the filename supplied by the user, or `None` if the user did not supply that option
- `args`, the list of positional arguments leftover after parsing options

This tutorial section only covers the four most important option attributes: *action*, *type*, *dest* (destination), and *help*. Of these, *action* is the most fundamental.

Understanding option actions

Actions tell *optparse* what to do when it encounters an option on the command line. There is a fixed set of actions hard-coded into *optparse*; adding new actions is an advanced topic covered in section *Extending optparse*. Most actions tell *optparse* to store a value in some variable—for example, take a string from the command line and store it in an attribute of `options`.

If you don't specify an option action, *optparse* defaults to `store`.

The store action

The most common option action is `store`, which tells *optparse* to take the next argument (or the remainder of the current argument), ensure that it is of the correct type, and store it to your chosen destination.

Por exemplo:

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

Now let's make up a fake command line and ask *optparse* to parse it:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

When *optparse* sees the option string `-f`, it consumes the next argument, `foo.txt`, and stores it in `options.filename`. So, after this call to `parse_args()`, `options.filename` is `"foo.txt"`.

Some other option types supported by *optparse* are `int` and `float`. Here's an option that expects an integer argument:

```
parser.add_option("-n", type="int", dest="num")
```

Note that this option has no long option string, which is perfectly acceptable. Also, there's no explicit action, since the default is `store`.

Let's parse another fake command-line. This time, we'll jam the option argument right up against the option: since `-n42` (one argument) is equivalent to `-n 42` (two arguments), the code

```
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

will print `42`.

If you don't specify a type, *optparse* assumes `string`. Combined with the fact that the default action is `store`, that means our first example can be a lot shorter:

```
parser.add_option("-f", "--file", dest="filename")
```

If you don't supply a destination, *optparse* figures out a sensible default from the option strings: if the first long option string is `--foo-bar`, then the default destination is `foo_bar`. If there are no long option strings, *optparse* looks at the first short option string: the default destination for `-f` is `f`.

optparse also includes the built-in `complex` type. Adding types is covered in section *Extending optparse*.

Handling boolean (flag) options

Flag options—set a variable to true or false when a particular option is seen—are quite common. *optparse* supports them with two separate actions, `store_true` and `store_false`. For example, you might have a `verbose` flag that is turned on with `-v` and off with `-q`:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Here we have two different options with the same destination, which is perfectly OK. (It just means you have to be a bit careful when setting default values—see below.)

When *optparse* encounters `-v` on the command line, it sets `options.verbose` to `True`; when it encounters `-q`, `options.verbose` is set to `False`.

Outras Ações

Some other actions supported by *optparse* are:

`"store_const"` armazena um valor constante

`"append"` append this option's argument to a list

`"count"` increment a counter by one

`"callback"` call a specified function

These are covered in section *Reference Guide*, Reference Guide and section *Option Callbacks*.

Default values

All of the above examples involve setting some variable (the "destination") when certain command-line options are seen. What happens if those options are never seen? Since we didn't supply any defaults, they are all set to `None`. This is usually fine, but sometimes you want more control. *optparse* lets you supply a default value for each destination, which is assigned before the command line is parsed.

First, consider the verbose/quiet example. If we want *optparse* to set `verbose` to `True` unless `-q` is seen, then we can do this:

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

Since default values apply to the *destination* rather than to any particular option, and these two options happen to have the same destination, this is exactly equivalent:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Consider this:

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Again, the default value for `verbose` will be `True`: the last default value supplied for any particular destination is the one that counts.

A clearer way to specify default values is the `set_defaults()` method of OptionParser, which you can call at any time before calling `parse_args()`:

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

As before, the last value specified for a given option destination is the one that counts. For clarity, try to use one method or the other of setting default values, not both.

Generating help

optparse's ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a *help* value for each option, and optionally a short usage message for your whole program. Here's an OptionParser populated with user-friendly (documented) options:

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE")
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, "
                       "or expert [default: %default]")
```

If *optparse* encounters either `-h` or `--help` on the command-line, or if you just call `parser.print_help()`, it prints the following to standard output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

(If the help output is triggered by a help option, *optparse* exits after printing the help text.)

There's a lot going on here to help *optparse* generate the best possible help message:

- the script defines its own usage message:

```
usage = "usage: %prog [options] arg1 arg2"
```

  optparse expands `%prog` in the usage string to the name of the current program, i.e. `os.path.basename(sys.argv[0])`. The expanded string is then printed before the detailed option help.

  If you don't supply a usage string, *optparse* uses a bland but sensible default: `"Usage: %prog [options]"`, which is fine if your script doesn't take any positional arguments.

- every option defines a help string, and doesn't worry about line-wrapping—*optparse* takes care of wrapping lines and making the help output look good.

- options that take a value indicate this fact in their automatically-generated help message, e.g. for the "mode" option:

```
-m MODE, --mode=MODE
```

  Here, "MODE" is called the meta-variable: it stands for the argument that the user is expected to supply to `-m`/`--mode`. By default, *optparse* converts the destination variable name to uppercase and uses that for the meta-variable. Sometimes, that's not what you want—for example, the `--filename` option explicitly sets `metavar="FILE"`, resulting in this automatically-generated option description:

```
-f FILE, --filename=FILE
```

  This is important for more than just saving space, though: the manually written help text uses the meta-variable `FILE` to clue the user in that there's a connection between the semi-formal syntax `-f FILE` and the informal semantic description "write output to FILE". This is a simple but effective way to make your help text a lot clearer and more useful for end users.

- options that have a default value can include `%default` in the help string—*optparse* will replace it with *str()* of the option's default value. If an option has no default value (or the default value is `None`), `%default` expands to `none`.

Grouping Options

When dealing with many options, it is convenient to group these options for better help output. An *OptionParser* can contain several option groups, each of which can contain several options.

An option group is obtained using the class *OptionGroup*:

class optparse.**OptionGroup**(*parser*, *title*, *description=None*)
  where

  - parser is the *OptionParser* instance the group will be inserted in to
  - title is the group title
  - description, optional, is a long description of the group

OptionGroup inherits from `OptionContainer` (like *OptionParser*) and so the `add_option()` method can be used to add an option to the group.

Once all the options are declared, using the *OptionParser* method `add_option_group()` the group is added to the previously defined parser.

Continuing with the parser defined in the previous section, adding an *OptionGroup* to a parser is easy:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

This would result in the following help output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.
```

A bit more complete example might involve using more than one group: still extending the previous example:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
                 help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
                 help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)
```

that results in the following output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or expert
                        [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.
```

(continua na próxima página)

(continuação da página anterior)

```
  -g                    Group option.

Debug Options:
  -d, --debug           Print debug information
  -s, --sql             Print all SQL statements executed
  -e                    Print every action done
```

Another interesting method, in particular when working programmatically with option groups is:

`OptionParser.`**`get_option_group`**(*opt_str*)
: Return the *OptionGroup* to which the short or long option string *opt_str* (e.g. `'-o'` or `'--option'`) belongs. If there's no such *OptionGroup*, return `None`.

Printing a version string

Similar to the brief usage string, *optparse* can also print a version string for your program. You have to supply the string as the `version` argument to OptionParser:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

`%prog` is expanded just like it is in `usage`. Apart from that, `version` can contain anything you like. When you supply it, *optparse* automatically adds a `--version` option to your parser. If it encounters this option on the command line, it expands your `version` string (by replacing `%prog`), prints it to stdout, and exits.

For example, if your script is called `/usr/bin/foo`:

```
$ /usr/bin/foo --version
foo 1.0
```

The following two methods can be used to print and get the `version` string:

`OptionParser.`**`print_version`**(*file=None*)
: Print the version message for the current program (`self.version`) to *file* (default stdout). As with *print_usage()*, any occurrence of `%prog` in `self.version` is replaced with the name of the current program. Does nothing if `self.version` is empty or undefined.

`OptionParser.`**`get_version`**()
: Same as *print_version()* but returns the version string instead of printing it.

How `optparse` handles errors

There are two broad classes of errors that *optparse* has to worry about: programmer errors and user errors. Programmer errors are usually erroneous calls to *OptionParser.add_option()*, e.g. invalid option strings, unknown option attributes, missing option attributes, etc. These are dealt with in the usual way: raise an exception (either `optparse.OptionError` or *TypeError*) and let the program crash.

Handling user errors is much more important, since they are guaranteed to happen no matter how stable your code is. *optparse* can automatically detect some user errors, such as bad option arguments (passing `-n 4x` where `-n` takes an integer argument), missing arguments (`-n` at the end of the command line, where `-n` takes an argument of any type). Also, you can call `OptionParser.error()` to signal an application-defined error condition:

```
(options, args) = parser.parse_args()
...
```

(continua na próxima página)

(continuação da página anterior)

```
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

In either case, *optparse* handles the error the same way: it prints the program's usage message and an error message to standard error and exits with error status 2.

Consider the first example above, where the user passes `4x` to an option that takes an integer:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

Or, where the user fails to pass a value at all:

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

optparse-generated error messages take care always to mention the option involved in the error; be sure to do the same when calling `OptionParser.error()` from your application code.

If *optparse*'s default error-handling behaviour does not suit your needs, you'll need to subclass OptionParser and override its `exit()` and/or `error()` methods.

Putting it all together

Here's what *optparse*-based scripts usually look like:

```
from optparse import OptionParser
...
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    ...
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print("reading %s..." % options.filename)
    ...

if __name__ == "__main__":
    main()
```

37.1.3 Reference Guide

Creating the parser

The first step in using *optparse* is to create an OptionParser instance.

class optparse.**OptionParser**(...)

The OptionParser constructor has no required arguments, but a number of optional keyword arguments. You should always pass them as keyword arguments, i.e. do not rely on the order in which the arguments are declared.

usage (default: **"%prog [options]"**) The usage summary to print when your program is run incorrectly or with a help option. When *optparse* prints the usage string, it expands `%prog` to `os.path.basename(sys.argv[0])` (or to `prog` if you passed that keyword argument). To suppress a usage message, pass the special value `optparse.SUPPRESS_USAGE`.

option_list (default: **[]**) A list of Option objects to populate the parser with. The options in `option_list` are added after any options in `standard_option_list` (a class attribute that may be set by OptionParser subclasses), but before any version or help options. Deprecated; use *add_option()* after creating the parser instead.

option_class (default: **optparse.Option**) Class to use when adding options to the parser in *add_option()*.

version (default: **None**) A version string to print when the user supplies a version option. If you supply a true value for `version`, *optparse* automatically adds a version option with the single option string `--version`. The substring `%prog` is expanded the same as for `usage`.

conflict_handler (default: **"error"**) Specifies what to do when options with conflicting option strings are added to the parser; see section *Conflicts between options*.

description (default: **None**) A paragraph of text giving a brief overview of your program. *optparse* reformats this paragraph to fit the current terminal width and prints it when the user requests help (after `usage`, but before the list of options).

formatter (default: **a new IndentedHelpFormatter**) An instance of optparse.HelpFormatter that will be used for printing help text. *optparse* provides two concrete classes for this purpose: IndentedHelpFormatter and TitledHelpFormatter.

add_help_option (default: **True**) If true, *optparse* will add a help option (with option strings `-h` and `--help`) to the parser.

prog The string to use when expanding `%prog` in `usage` and `version` instead of `os.path.basename(sys.argv[0])`.

epilog (default: **None**) A paragraph of help text to print after the option help.

Populating the parser

There are several ways to populate the parser with options. The preferred way is by using *OptionParser.add_option()*, as shown in section *Tutorial*. `add_option()` can be called in one of two ways:

- pass it an Option instance (as returned by `make_option()`)
- pass it any combination of positional and keyword arguments that are acceptable to `make_option()` (i.e., to the Option constructor), and it will create the Option instance for you

The other alternative is to pass a list of pre-constructed Option instances to the OptionParser constructor, as in:

```
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
    ]
parser = OptionParser(option_list=option_list)
```

(`make_option()` is a factory function for creating Option instances; currently it is an alias for the Option constructor. A future version of *optparse* may split Option into several classes, and `make_option()` will pick the right class to instantiate. Do not instantiate Option directly.)

Defining options

Each Option instance represents a set of synonymous command-line option strings, e.g. `-f` and `--file`. You can specify any number of short or long option strings, but you must specify at least one overall option string.

The canonical way to create an `Option` instance is with the `add_option()` method of *OptionParser*.

`OptionParser.`**`add_option`**`(`*option*`)`
`OptionParser.`**`add_option`**`(`**opt_str*, *attr=value*, *...*`)`

To define an option with only a short option string:

```
parser.add_option("-f", attr=value, ...)
```

And to define an option with only a long option string:

```
parser.add_option("--foo", attr=value, ...)
```

The keyword arguments define attributes of the new Option object. The most important option attribute is *action*, and it largely determines which other attributes are relevant or required. If you pass irrelevant option attributes, or fail to pass required ones, *optparse* raises an `OptionError` exception explaining your mistake.

An option's *action* determines what *optparse* does when it encounters this option on the command-line. The standard option actions hard-coded into *optparse* are:

`"store"` store this option's argument (default)

`"store_const"` armazena um valor constante

`"store_true"` store `True`

`"store_false"` store `False`

`"append"` append this option's argument to a list

`"append_const"` append a constant value to a list

`"count"` increment a counter by one

`"callback"` call a specified function

`"help"` print a usage message including all options and the documentation for them

(If you don't supply an action, the default is `"store"`. For this action, you may also supply *type* and *dest* option attributes; see *Standard option actions*.)

As you can see, most actions involve storing or updating a value somewhere. *optparse* always creates a special object for this, conventionally called `options` (it happens to be an instance of `optparse.Values`). Option arguments (and various other values) are stored as attributes of this object, according to the *dest* (destination) option attribute.

For example, when you call

```
parser.parse_args()
```

one of the first things *optparse* does is create the `options` object:

```
options = Values()
```

If one of the options in this parser is defined with

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

and the command-line being parsed includes any of the following:

```
-ffoo
-f foo
--file=foo
--file foo
```

then *optparse*, on seeing this option, will do the equivalent of

```
options.filename = "foo"
```

The *type* and *dest* option attributes are almost as important as *action*, but *action* is the only one that makes sense for *all* options.

Option attributes

The following option attributes may be passed as keyword arguments to *OptionParser.add_option()*. If you pass an option attribute that is not relevant to a particular option, or fail to pass a required option attribute, *optparse* raises `OptionError`.

`Option.`**`action`**
(default: `"store"`)

Determines *optparse*'s behaviour when this option is seen on the command line; the available options are documented *here*.

`Option.`**`type`**
(default: `"string"`)

The argument type expected by this option (e.g., `"string"` or `"int"`); the available option types are documented *here*.

`Option.`**`dest`**
(default: derived from option strings)

If the option's action implies writing or modifying a value somewhere, this tells *optparse* where to write it: *dest* names an attribute of the `options` object that *optparse* builds as it parses the command line.

`Option.`**`default`**
The value to use for this option's destination if the option is not seen on the command line. See also *OptionParser.set_defaults()*.

`Option.`**`nargs`**
(default: 1)

How many arguments of type *type* should be consumed when this option is seen. If > 1, *optparse* will store a tuple of values to *dest*.

Option.**const**
For actions that store a constant value, the constant value to store.

Option.**choices**
For options of type "choice", the list of strings the user may choose from.

Option.**callback**
For options with action "callback", the callable to call when this option is seen. See section *Option Callbacks* for detail on the arguments passed to the callable.

Option.**callback_args**
Option.**callback_kwargs**
Additional positional and keyword arguments to pass to callback after the four standard callback arguments.

Option.**help**
Help text to print for this option when listing all available options after the user supplies a *help* option (such as --help). If no help text is supplied, the option will be listed without help text. To hide this option, use the special value optparse.SUPPRESS_HELP.

Option.**metavar**
(default: derived from option strings)

Stand-in for the option argument(s) to use when printing help text. See section *Tutorial* for an example.

Standard option actions

The various option actions all have slightly different requirements and effects. Most actions have several relevant option attributes which you may specify to guide *optparse*'s behaviour; a few have required attributes, which you must specify for any option using that action.

- "store" [relevant: *type*, *dest*, *nargs*, *choices*]

  The option must be followed by an argument, which is converted to a value according to *type* and stored in *dest*. If *nargs* > 1, multiple arguments will be consumed from the command line; all will be converted according to *type* and stored to *dest* as a tuple. See the *Standard option types* section.

  If *choices* is supplied (a list or tuple of strings), the type defaults to "choice".

  If *type* is not supplied, it defaults to "string".

  If *dest* is not supplied, *optparse* derives a destination from the first long option string (e.g., --foo-bar implies foo_bar). If there are no long option strings, *optparse* derives a destination from the first short option string (e.g., -f implies f).

  Exemplo:

```
parser.add_option("-f")
parser.add_option("-p", type="float", nargs=3, dest="point")
```

  As it parses the command line

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

  optparse will set

```
options.f = "foo.txt"
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"
```

- `"store_const"` [required: *const*; relevant: *dest*]

  The value *const* is stored in *dest*.

  Exemplo:

```
parser.add_option("-q", "--quiet",
                  action="store_const", const=0, dest="verbose")
parser.add_option("-v", "--verbose",
                  action="store_const", const=1, dest="verbose")
parser.add_option("--noisy",
                  action="store_const", const=2, dest="verbose")
```

  If `--noisy` is seen, *optparse* will set

```
options.verbose = 2
```

- `"store_true"` [relevant: *dest*]

  A special case of `"store_const"` that stores `True` to *dest*.

- `"store_false"` [relevant: *dest*]

  Like `"store_true"`, but stores `False`.

  Exemplo:

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- `"append"` [relevant: *type*, *dest*, *nargs*, *choices*]

  The option must be followed by an argument, which is appended to the list in *dest*. If no default value for *dest* is supplied, an empty list is automatically created when *optparse* first encounters this option on the command-line. If *nargs* > 1, multiple arguments are consumed, and a tuple of length *nargs* is appended to *dest*.

  The defaults for *type* and *dest* are the same as for the `"store"` action.

  Exemplo:

```
parser.add_option("-t", "--tracks", action="append", type="int")
```

  If `-t3` is seen on the command-line, *optparse* does the equivalent of:

```
options.tracks = []
options.tracks.append(int("3"))
```

  If, a little later on, `--tracks=4` is seen, it does:

```
options.tracks.append(int("4"))
```

  The `append` action calls the `append` method on the current value of the option. This means that any default value specified must have an `append` method. It also means that if the default value is non-empty, the default elements will be present in the parsed value for the option, with any values from the command line appended after those default values:

```
>>> parser.add_option("--files", action="append", default=['~/.mypkg/defaults'])
>>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
>>> opts.files
['~/.mypkg/defaults', 'overrides.mypkg']
```

- `"append_const"` [required: *const*; relevant: *dest*]

  Like `"store_const"`, but the value *const* is appended to *dest*; as with `"append"`, *dest* defaults to `None`, and an empty list is automatically created the first time the option is encountered.

- `"count"` [relevant: *dest*]

  Increment the integer stored at *dest*. If no default value is supplied, *dest* is set to zero before being incremented the first time.

  Exemplo:

```
parser.add_option("-v", action="count", dest="verbosity")
```

  The first time `-v` is seen on the command line, *optparse* does the equivalent of:

```
options.verbosity = 0
options.verbosity += 1
```

  Every subsequent occurrence of `-v` results in

```
options.verbosity += 1
```

- `"callback"` [required: *callback*; relevant: *type*, *nargs*, *callback_args*, *callback_kwargs*]

  Call the function specified by *callback*, which is called as

```
func(option, opt_str, value, parser, *args, **kwargs)
```

  See section *Option Callbacks* for more detail.

- `"help"`

  Prints a complete help message for all the options in the current option parser. The help message is constructed from the `usage` string passed to OptionParser's constructor and the *help* string passed to every option.

  If no *help* string is supplied for an option, it will still be listed in the help message. To omit an option entirely, use the special value `optparse.SUPPRESS_HELP`.

  optparse automatically adds a *help* option to all OptionParsers, so you do not normally need to create one.

  Exemplo:

```
from optparse import OptionParser, SUPPRESS_HELP

# usually, a help option is added automatically, but that can
# be suppressed using the add_help_option argument
parser = OptionParser(add_help_option=False)

parser.add_option("-h", "--help", action="help")
parser.add_option("-v", action="store_true", dest="verbose",
                  help="Be moderately verbose")
parser.add_option("--file", dest="filename",
                  help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)
```

  If *optparse* sees either `-h` or `--help` on the command line, it will print something like the following help message to stdout (assuming `sys.argv[0]` is `"foo.py"`):

```
Usage: foo.py [options]

Options:
  -h, --help        Show this help message and exit
  -v                Be moderately verbose
  --file=FILENAME   Input file to read data from
```

After printing the help message, *optparse* terminates your process with `sys.exit(0)`.

- `"version"`

  Prints the version number supplied to the OptionParser to stdout and exits. The version number is actually formatted and printed by the `print_version()` method of OptionParser. Generally only relevant if the `version` argument is supplied to the OptionParser constructor. As with *help* options, you will rarely create `version` options, since *optparse* automatically adds them when needed.

Standard option types

optparse has five built-in option types: `"string"`, `"int"`, `"choice"`, `"float"` and `"complex"`. If you need to add new option types, see section *Extending optparse*.

Arguments to string options are not checked or converted in any way: the text on the command line is stored in the destination (or passed to the callback) as-is.

Integer arguments (type `"int"`) are parsed as follows:

- if the number starts with `0x`, it is parsed as a hexadecimal number
- if the number starts with `0`, it is parsed as an octal number
- if the number starts with `0b`, it is parsed as a binary number
- otherwise, the number is parsed as a decimal number

The conversion is done by calling *int()* with the appropriate base (2, 8, 10, or 16). If this fails, so will *optparse*, although with a more useful error message.

`"float"` and `"complex"` option arguments are converted directly with *float()* and *complex()*, with similar error-handling.

`"choice"` options are a subtype of `"string"` options. The *choices* option attribute (a sequence of strings) defines the set of allowed option arguments. `optparse.check_choice()` compares user-supplied option arguments against this master list and raises `OptionValueError` if an invalid string is given.

Análise de argumentos

The whole point of creating and populating an OptionParser is to call its `parse_args()` method:

```
(options, args) = parser.parse_args(args=None, values=None)
```

where the input parameters are

args the list of arguments to process (default: `sys.argv[1:]`)

values an `optparse.Values` object to store option arguments in (default: a new instance of `Values`) – if you give an existing object, the option defaults will not be initialized on it

and the return values are

options the same object that was passed in as `values`, or the optparse.Values instance created by *optparse*

args the leftover positional arguments after all options have been processed

The most common usage is to supply neither keyword argument. If you supply `values`, it will be modified with repeated *setattr()* calls (roughly one for every option argument stored to an option destination) and returned by `parse_args()`.

If `parse_args()` encounters any errors in the argument list, it calls the OptionParser's `error()` method with an appropriate end-user error message. This ultimately terminates your process with an exit status of 2 (the traditional Unix exit status for command-line errors).

Querying and manipulating your option parser

The default behavior of the option parser can be customized slightly, and you can also poke around your option parser and see what's there. OptionParser provides several methods to help you out:

`OptionParser.`**`disable_interspersed_args`**`()`
Set parsing to stop on the first non-option. For example, if `-a` and `-b` are both simple options that take no arguments, *optparse* normally accepts this syntax:

```
prog -a arg1 -b arg2
```

and treats it as equivalent to

```
prog -a -b arg1 arg2
```

To disable this feature, call *disable_interspersed_args()*. This restores traditional Unix syntax, where option parsing stops with the first non-option argument.

Use this if you have a command processor which runs another command which has options of its own and you want to make sure these options don't get confused. For example, each command might have a different set of options.

`OptionParser.`**`enable_interspersed_args`**`()`
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior.

`OptionParser.`**`get_option`**`(`*opt_str*`)`
Returns the Option instance with the option string *opt_str*, or `None` if no options have that option string.

`OptionParser.`**`has_option`**`(`*opt_str*`)`
Return `True` if the OptionParser has an option with option string *opt_str* (e.g., `-q` or `--verbose`).

`OptionParser.`**`remove_option`**`(`*opt_str*`)`
If the *OptionParser* has an option corresponding to *opt_str*, that option is removed. If that option provided any other option strings, all of those option strings become invalid. If *opt_str* does not occur in any option belonging to this *OptionParser*, raises *ValueError*.

Conflicts between options

If you're not careful, it's easy to define options with conflicting option strings:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(This is particularly true if you've defined your own OptionParser subclass with some standard options.)

Every time you add an option, *optparse* checks for conflicts with existing options. If it finds any, it invokes the current conflict-handling mechanism. You can set the conflict-handling mechanism either in the constructor:

```
parser = OptionParser(..., conflict_handler=handler)
```

or with a separate call:

```
parser.set_conflict_handler(handler)
```

The available conflict handlers are:

"error" (default) assume option conflicts are a programming error and raise OptionConflictError

"resolve" resolve option conflicts intelligently (see below)

As an example, let's define an *OptionParser* that resolves conflicts intelligently and add conflicting options to it:

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

At this point, *optparse* detects that a previously-added option is already using the `-n` option string. Since `conflict_handler` is `"resolve"`, it resolves the situation by removing `-n` from the earlier option's list of option strings. Now `--dry-run` is the only way for the user to activate that option. If the user asks for help, the help message will reflect that:

```
Options:
  --dry-run     do no harm
  ...
  -n, --noisy   be noisy
```

It's possible to whittle away the option strings for a previously-added option until there are none left, and the user has no way of invoking that option from the command-line. In that case, *optparse* removes that option completely, so it doesn't show up in help text or anywhere else. Carrying on with our existing OptionParser:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

At this point, the original `-n`/`--dry-run` option is no longer accessible, so *optparse* removes it, leaving this help text:

```
Options:
  ...
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

Cleanup

OptionParser instances have several cyclic references. This should not be a problem for Python's garbage collector, but you may wish to break the cyclic references explicitly by calling `destroy()` on your OptionParser once you are done with it. This is particularly useful in long-running applications where large object graphs are reachable from your OptionParser.

Other methods

OptionParser supports several other public methods:

`OptionParser.`**`set_usage`**(*usage*)
Set the usage string according to the rules described above for the `usage` constructor keyword argument. Passing `None` sets the default usage string; use `optparse.SUPPRESS_USAGE` to suppress a usage message.

`OptionParser.`**`print_usage`**(*file=None*)
Print the usage message for the current program (`self.usage`) to *file* (default stdout). Any occurrence of the string `%prog` in `self.usage` is replaced with the name of the current program. Does nothing if `self.usage` is empty or not defined.

`OptionParser.`**`get_usage`**()
Same as *`print_usage()`* but returns the usage string instead of printing it.

`OptionParser.`**`set_defaults`**(*dest=value, ...*)
Set default values for several option destinations at once. Using *`set_defaults()`* is the preferred way to set default values for options, since multiple options can share the same destination. For example, if several "mode" options all set the same destination, any one of them can set the default, and the last one wins:

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")    # overridden below
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")  # overrides above setting
```

To avoid this confusion, use *`set_defaults()`*:

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

37.1.4 Option Callbacks

When *`optparse`*'s built-in actions and types aren't quite enough for your needs, you have two choices: extend *`optparse`* or define a callback option. Extending *`optparse`* is more general, but overkill for a lot of simple cases. Quite often a simple callback is all you need.

There are two steps to defining a callback option:

- define the option itself using the `"callback"` action
- write the callback; this is a function (or method) that takes at least four arguments, as described below

Defining a callback option

As always, the easiest way to define a callback option is by using the *OptionParser.add_option()* method. Apart from *action*, the only option attribute you must specify is `callback`, the function to call:

```
parser.add_option("-c", action="callback", callback=my_callback)
```

`callback` is a function (or other callable object), so you must have already defined `my_callback()` when you create this callback option. In this simple case, *optparse* doesn't even know if `-c` takes any arguments, which usually means that the option takes no arguments—the mere presence of `-c` on the command-line is all it needs to know. In some circumstances, though, you might want your callback to consume an arbitrary number of command-line arguments. This is where writing callbacks gets tricky; it's covered later in this section.

optparse always passes four particular arguments to your callback, and it will only pass additional arguments if you specify them via *callback_args* and *callback_kwargs*. Thus, the minimal callback function signature is:

```
def my_callback(option, opt, value, parser):
```

The four arguments to a callback are described below.

There are several other option attributes that you can supply when you define a callback option:

type has its usual meaning: as with the `"store"` or `"append"` actions, it instructs *optparse* to consume one argument and convert it to *type*. Rather than storing the converted value(s) anywhere, though, *optparse* passes it to your callback function.

nargs also has its usual meaning: if it is supplied and > 1, *optparse* will consume *nargs* arguments, each of which must be convertible to *type*. It then passes a tuple of converted values to your callback.

callback_args a tuple of extra positional arguments to pass to the callback

callback_kwargs a dictionary of extra keyword arguments to pass to the callback

How callbacks are called

All callbacks are called as follows:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

where

`option` is the Option instance that's calling the callback

`opt_str` is the option string seen on the command-line that's triggering the callback. (If an abbreviated long option was used, `opt_str` will be the full, canonical option string—e.g. if the user puts `--foo` on the command-line as an abbreviation for `--foobar`, then `opt_str` will be `"--foobar"`.)

`value` is the argument to this option seen on the command-line. *optparse* will only expect an argument if *type* is set; the type of `value` will be the type implied by the option's type. If *type* for this option is `None` (no argument expected), then `value` will be `None`. If *nargs* > 1, `value` will be a tuple of values of the appropriate type.

`parser` is the OptionParser instance driving the whole thing, mainly useful because you can access some other interesting data through its instance attributes:

`parser.largs` the current list of leftover arguments, ie. arguments that have been consumed but are neither options nor option arguments. Feel free to modify `parser.largs`, e.g. by adding more arguments to it. (This list will become `args`, the second return value of `parse_args()`.)

parser.rargs the current list of remaining arguments, ie. with `opt_str` and `value` (if applicable) removed, and only the arguments following them still there. Feel free to modify `parser.rargs`, e.g. by consuming more arguments.

parser.values the object where option values are by default stored (an instance of optparse.OptionValues). This lets callbacks use the same mechanism as the rest of *optparse* for storing option values; you don't need to mess around with globals or closures. You can also access or modify the value(s) of any options already encountered on the command-line.

args is a tuple of arbitrary positional arguments supplied via the *callback_args* option attribute.

kwargs is a dictionary of arbitrary keyword arguments supplied via *callback_kwargs*.

Raising errors in a callback

The callback function should raise `OptionValueError` if there are any problems with the option or its argument(s). *optparse* catches this and terminates the program, printing the error message you supply to stderr. Your message should be clear, concise, accurate, and mention the option at fault. Otherwise, the user will have a hard time figuring out what they did wrong.

Callback example 1: trivial callback

Here's an example of a callback option that takes no arguments, and simply records that the option was seen:

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

Of course, you could do that with the `"store_true"` action.

Callback example 2: check option order

Here's a slightly more interesting example: record the fact that `-a` is seen, but blow up if it comes after `-b` in the command-line.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

Callback example 3: check option order (generalized)

If you want to re-use this callback for several similar options (set a flag, but blow up if `-b` has already been seen), it needs a bit of work: the error message and the flag that it sets must be generalized.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

Callback example 4: check arbitrary condition

Of course, you could put any condition in there—you're not limited to checking the values of already-defined options. For example, if you have options that should not be called when the moon is full, all you have to do is this:

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full"
                               % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

(The definition of `is_moon_full()` is left as an exercise for the reader.)

Callback example 5: fixed arguments

Things get slightly more interesting when you define callback options that take a fixed number of arguments. Specifying that a callback option takes arguments is similar to defining a `"store"` or `"append"` option: if you define *type*, then the option takes one argument that must be convertible to that type; if you further define *nargs*, then the option takes *nargs* arguments.

Here's an example that just emulates the standard `"store"` action:

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

Note that *optparse* takes care of consuming 3 arguments and converting them to integers for you; all you have to do is store them. (Or whatever; obviously you don't need a callback for this example.)

Callback example 6: variable arguments

Things get hairy when you want an option to take a variable number of arguments. For this case, you must write a callback, as *optparse* doesn't provide any built-in capabilities for it. And you have to deal with certain intricacies of conventional Unix command-line parsing that *optparse* normally handles for you. In particular, callbacks should implement the conventional rules for bare `--` and `-` arguments:

- either `--` or `-` can be option arguments
- bare `--` (if not the argument to some option): halt command-line processing and discard the `--`
- bare `-` (if not the argument to some option): halt command-line processing but keep the `-` (append it to `parser.largs`)

If you want an option that takes a variable number of arguments, there are several subtle, tricky issues to worry about. The exact implementation you choose will be based on which trade-offs you're willing to make for your application (which is why *optparse* doesn't support this sort of thing directly).

Nevertheless, here's a stab at a callback for an option with variable arguments:

```
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []

    def floatable(str):
        try:
            float(str)
            return True
        except ValueError:
            return False

    for arg in parser.rargs:
        # stop on --foo like options
        if arg[:2] == "--" and len(arg) > 2:
            break
        # stop on -a, but not on -3 or -3.0
        if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
            break
        value.append(arg)

    del parser.rargs[:len(value)]
    setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback", dest="vararg_attr",
                  action="callback", callback=vararg_callback)
```

37.1.5 Extending `optparse`

Since the two major controlling factors in how *optparse* interprets command-line options are the action and type of each option, the most likely direction of extension is to add new actions and new types.

Adding new types

To add new types, you need to define your own subclass of *optparse*'s `Option` class. This class has a couple of attributes that define *optparse*'s types: *TYPES* and *TYPE_CHECKER*.

`Option.`**`TYPES`**
A tuple of type names; in your subclass, simply define a new tuple *TYPES* that builds on the standard one.

`Option.`**`TYPE_CHECKER`**
A dictionary mapping type names to type-checking functions. A type-checking function has the following signature:

```
def check_mytype(option, opt, value)
```

where `option` is an `Option` instance, `opt` is an option string (e.g., `-f`), and `value` is the string from the command line that must be checked and converted to your desired type. `check_mytype()` should return an object of the hypothetical type `mytype`. The value returned by a type-checking function will wind up in the OptionValues instance returned by `OptionParser.parse_args()`, or be passed to a callback as the `value` parameter.

Your type-checking function should raise `OptionValueError` if it encounters any problems. `OptionValueError` takes a single string argument, which is passed as-is to *OptionParser*'s `error()` method, which in turn prepends the program name and the string `"error:"` and prints everything to stderr before terminating the process.

Here's a silly example that demonstrates adding a `"complex"` option type to parse Python-style complex numbers on the command line. (This is even sillier than it used to be, because *optparse* 1.3 added built-in support for complex numbers, but never mind.)

First, the necessary imports:

```
from copy import copy
from optparse import Option, OptionValueError
```

You need to define your type-checker first, since it's referred to later (in the *TYPE_CHECKER* class attribute of your Option subclass):

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

Finally, the Option subclass:

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(If we didn't make a *copy()* of *Option.TYPE_CHECKER*, we would end up modifying the *TYPE_CHECKER* attribute of *optparse*'s Option class. This being Python, nothing stops you from doing that except good manners and common sense.)

That's it! Now you can write a script that uses the new option type just like any other *optparse*-based script, except you have to instruct your OptionParser to use MyOption instead of Option:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

Alternately, you can build your own option list and pass it to OptionParser; if you don't use `add_option()` in the above way, you don't need to tell OptionParser which option class to use:

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

Adding new actions

Adding new actions is a bit trickier, because you have to understand that *optparse* has a couple of classifications for actions:

"store" actions actions that result in *optparse* storing a value to an attribute of the current OptionValues instance; these options require a *dest* attribute to be supplied to the Option constructor.

"typed" actions actions that take a value from the command line and expect it to be of a certain type; or rather, a string that can be converted to a certain type. These options require a *type* attribute to the Option constructor.

These are overlapping sets: some default "store" actions are `"store"`, `"store_const"`, `"append"`, and `"count"`, while the default "typed" actions are `"store"`, `"append"`, and `"callback"`.

When you add an action, you need to categorize it by listing it in at least one of the following class attributes of Option (all are lists of strings):

`Option.`**ACTIONS**
All actions must be listed in ACTIONS.

`Option.`**STORE_ACTIONS**
"store" actions are additionally listed here.

`Option.`**TYPED_ACTIONS**
"typed" actions are additionally listed here.

`Option.`**ALWAYS_TYPED_ACTIONS**
Actions that always take a type (i.e. whose options always take a value) are additionally listed here. The only effect of this is that *optparse* assigns the default type, `"string"`, to options with no explicit type whose action is listed in *ALWAYS_TYPED_ACTIONS*.

In order to actually implement your new action, you must override Option's `take_action()` method and add a case that recognizes your action.

For example, let's add an `"extend"` action. This is similar to the standard `"append"` action, but instead of taking a single value from the command-line and appending it to an existing list, `"extend"` will take multiple values in a single comma-delimited string, and extend an existing list with them. That is, if `--names` is an `"extend"` option of type `"string"`, the command line

```
--names=foo,bar --names blah --names ding,dong
```

would result in a list

```
["foo", "bar", "blah", "ding", "dong"]
```

Again we define a subclass of Option:

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
```

(continua na próxima página)

(continuação da página anterior)

```
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

Features of note:

- `"extend"` both expects a value on the command-line and stores that value somewhere, so it goes in both *STORE_ACTIONS* and *TYPED_ACTIONS*.
- to ensure that *optparse* assigns the default type of `"string"` to `"extend"` actions, we put the `"extend"` action in *ALWAYS_TYPED_ACTIONS* as well.
- `MyOption.take_action()` implements just this one new action, and passes control back to `Option.take_action()` for the standard *optparse* actions.
- `values` is an instance of the optparse_parser.Values class, which provides the very useful `ensure_value()` method. `ensure_value()` is essentially *getattr()* with a safety valve; it is called as

  ```
  values.ensure_value(attr, value)
  ```

  If the `attr` attribute of `values` doesn't exist or is `None`, then ensure_value() first sets it to `value`, and then returns 'value. This is very handy for actions like `"extend"`, `"append"`, and `"count"`, all of which accumulate data in a variable and expect that variable to be of a certain type (a list for the first two, an integer for the latter). Using `ensure_value()` means that scripts using your action don't have to worry about setting a default value for the option destinations in question; they can just leave the default as `None` and `ensure_value()` will take care of getting it right when it's needed.

37.2 `imp` — Access the import internals

Código Fonte: Lib/imp.py

Obsoleto desde a versão 3.4: The *imp* module is deprecated in favor of *importlib*.

This module provides an interface to the mechanisms used to implement the `import` statement. It defines the following constants and functions:

`imp.`**`get_magic`**`()`
: Return the magic string value used to recognize byte-compiled code files (`.pyc` files). (This value may be different for each Python version.)

  Obsoleto desde a versão 3.4: Use *importlib.util.MAGIC_NUMBER* instead.

`imp.`**`get_suffixes`**`()`
: Return a list of 3-element tuples, each describing a particular type of module. Each triple has the form `(suffix, mode, type)`, where *suffix* is a string to be appended to the module name to form the filename to search for, *mode* is the mode string to pass to the built-in *open()* function to open the file (this can be `'r'` for text files or `'rb'` for binary files), and *type* is the file type, which has one of the values *PY_SOURCE*, *PY_COMPILED*, or *C_EXTENSION*, described below.

  Obsoleto desde a versão 3.3: Use the constants defined on *importlib.machinery* instead.

imp.**find_module**(*name*[, *path*])

Try to find the module *name*. If *path* is omitted or `None`, the list of directory names given by `sys.path` is searched, but first a few special places are searched: the function tries to find a built-in module with the given name (*C_BUILTIN*), then a frozen module (*PY_FROZEN*), and on some systems some other places are looked in as well (on Windows, it looks in the registry which may point to a specific file).

Otherwise, *path* must be a list of directory names; each directory is searched for files with any of the suffixes returned by *get_suffixes()* above. Invalid names in the list are silently ignored (but all list items must be strings).

If search is successful, the return value is a 3-element tuple `(file, pathname, description)`:

file is an open *file object* positioned at the beginning, *pathname* is the pathname of the file found, and *description* is a 3-element tuple as contained in the list returned by *get_suffixes()* describing the kind of module found.

If the module is built-in or frozen then *file* and *pathname* are both `None` and the *description* tuple contains empty strings for its suffix and mode; the module type is indicated as given in parentheses above. If the search is unsuccessful, *ImportError* is raised. Other exceptions indicate problems with the arguments or environment.

If the module is a package, *file* is `None`, *pathname* is the package path and the last item in the *description* tuple is *PKG_DIRECTORY*.

This function does not handle hierarchical module names (names containing dots). In order to find *P.M*, that is, submodule *M* of package *P*, use *find_module()* and *load_module()* to find and load package *P*, and then use *find_module()* with the *path* argument set to `P.__path__`. When *P* itself has a dotted name, apply this recipe recursively.

Obsoleto desde a versão 3.3: Use *importlib.util.find_spec()* instead unless Python 3.3 compatibility is required, in which case use *importlib.find_loader()*. For example usage of the former case, see the *Exemplos* section of the *importlib* documentation.

imp.**load_module**(*name*, *file*, *pathname*, *description*)

Load a module that was previously found by *find_module()* (or by an otherwise conducted search yielding compatible results). This function does more than importing the module: if the module was already imported, it will reload the module! The *name* argument indicates the full module name (including the package name, if this is a submodule of a package). The *file* argument is an open file, and *pathname* is the corresponding file name; these can be `None` and `''`, respectively, when the module is a package or not being loaded from a file. The *description* argument is a tuple, as would be returned by *get_suffixes()*, describing what kind of module must be loaded.

If the load is successful, the return value is the module object; otherwise, an exception (usually *ImportError*) is raised.

Important: the caller is responsible for closing the *file* argument, if it was not `None`, even when an exception is raised. This is best done using a `try` ... `finally` statement.

Obsoleto desde a versão 3.3: If previously used in conjunction with *imp.find_module()* then consider using *importlib.import_module()*, otherwise use the loader returned by the replacement you chose for *imp.find_module()*. If you called *imp.load_module()* and related functions directly with file path arguments then use a combination of *importlib.util.spec_from_file_location()* and *importlib.util.module_from_spec()*. See the *Exemplos* section of the *importlib* documentation for details of the various approaches.

imp.**new_module**(*name*)

Return a new empty module object called *name*. This object is *not* inserted in `sys.modules`.

Obsoleto desde a versão 3.4: Use *importlib.util.module_from_spec()* instead.

imp.**reload**(*module*)

Reload a previously imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try

out the new version without leaving the Python interpreter. The return value is the module object (the same as the *module* argument).

When `reload(module)` is executed:

- Python modules' code is recompiled and the module-level code reexecuted, defining a new set of objects which are bound to names in the module's dictionary. The `init` function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects — with a `try` statement it can test for the table's presence and skip its initialization if desired:

```
try:
    cache
except NameError:
    cache = {}
```

It is legal though generally not very useful to reload built-in or dynamically loaded modules, except for *sys*, *__main__* and *builtins*. In many cases, however, extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from` … `import` …, calling *reload()* for the other module does not redefine the objects imported from it — one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (*module*.*name*) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

Alterado na versão 3.3: Relies on both `__name__` and `__loader__` being defined on the module being reloaded instead of just `__name__`.

Obsoleto desde a versão 3.4: Use *importlib.reload()* instead.

The following functions are conveniences for handling **PEP 3147** byte-compiled file paths.

Novo na versão 3.2.

`imp.`**`cache_from_source`**(*path*, *debug_override=None*)

Return the **PEP 3147** path to the byte-compiled file associated with the source *path*. For example, if *path* is `/foo/bar/baz.py` the return value would be `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. The `cpython-32` string comes from the current magic tag (see *get_tag()*; if `sys.implementation.cache_tag` is not defined then *NotImplementedError* will be raised). By passing in `True` or `False` for *debug_override* you can override the system's value for `__debug__`, leading to optimized bytecode.

path need not exist.

Alterado na versão 3.3: If `sys.implementation.cache_tag` is `None`, then *NotImplementedError* is raised.

Obsoleto desde a versão 3.4: Use *importlib.util.cache_from_source()* instead.

Alterado na versão 3.5: The *debug_override* parameter no longer creates a `.pyo` file.

imp.**source_from_cache**(*path*)
Given the *path* to a **PEP 3147** file name, return the associated source code file path. For example, if *path* is `/foo/bar/__pycache__/baz.cpython-32.pyc` the returned path would be `/foo/bar/baz.py`. *path* need not exist, however if it does not conform to **PEP 3147** format, a *ValueError* is raised. If `sys.implementation.cache_tag` is not defined, *NotImplementedError* is raised.

Alterado na versão 3.3: Raise *NotImplementedError* when `sys.implementation.cache_tag` is not defined.

Obsoleto desde a versão 3.4: Use *importlib.util.source_from_cache()* instead.

imp.**get_tag**()
Return the **PEP 3147** magic tag string matching this version of Python's magic number, as returned by *get_magic()*.

Obsoleto desde a versão 3.4: Use `sys.implementation.cache_tag` directly starting in Python 3.3.

The following functions help interact with the import system's internal locking mechanism. Locking semantics of imports are an implementation detail which may vary from release to release. However, Python ensures that circular imports work without any deadlocks.

imp.**lock_held**()
Return `True` if the global import lock is currently held, else `False`. On platforms without threads, always return `False`.

On platforms with threads, a thread executing an import first holds a global import lock, then sets up a per-module lock for the rest of the import. This blocks other threads from importing the same module until the original import completes, preventing other threads from seeing incomplete module objects constructed by the original thread. An exception is made for circular imports, which by construction have to expose an incomplete module object at some point.

Alterado na versão 3.3: The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

Obsoleto desde a versão 3.4.

imp.**acquire_lock**()
Acquire the interpreter's global import lock for the current thread. This lock should be used by import hooks to ensure thread-safety when importing modules.

Once a thread has acquired the import lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

On platforms without threads, this function does nothing.

Alterado na versão 3.3: The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

Obsoleto desde a versão 3.4.

imp.**release_lock**()
Release the interpreter's global import lock. On platforms without threads, this function does nothing.

Alterado na versão 3.3: The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

Obsoleto desde a versão 3.4.

The following constants with integer values, defined in this module, are used to indicate the search result of *find_module()*.

imp.**PY_SOURCE**
The module was found as a source file.

Obsoleto desde a versão 3.3.

imp.**PY_COMPILED**
The module was found as a compiled code object file.

Obsoleto desde a versão 3.3.

imp.**C_EXTENSION**
The module was found as dynamically loadable shared library.

Obsoleto desde a versão 3.3.

imp.**PKG_DIRECTORY**
The module was found as a package directory.

Obsoleto desde a versão 3.3.

imp.**C_BUILTIN**
The module was found as a built-in module.

Obsoleto desde a versão 3.3.

imp.**PY_FROZEN**
The module was found as a frozen module.

Obsoleto desde a versão 3.3.

class imp.**NullImporter**(*path_string*)
The *NullImporter* type is a **PEP 302** import hook that handles non-directory path strings by failing to find any modules. Calling this type with an existing directory or empty string raises *ImportError*. Otherwise, a *NullImporter* instance is returned.

Instances have only one method:

find_module(*fullname*[, *path*])
This method always returns None, indicating that the requested module could not be found.

Alterado na versão 3.3: None is inserted into sys.path_importer_cache instead of an instance of *NullImporter*.

Obsoleto desde a versão 3.4: Insert None into sys.path_importer_cache instead.

37.2.1 Exemplos

The following function emulates what was the standard import statement up to Python 1.4 (no hierarchical module names). (This *implementation* wouldn't work in that version, since *find_module()* has been extended and *load_module()* has been added in 1.4.)

```
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
    # Fast path: see if the module has already been imported.
    try:
        return sys.modules[name]
    except KeyError:
        pass
```

(continua na próxima página)

(continuação da página anterior)

```
    # If any of the following calls raises an exception,
    # there's a problem we can't handle -- let the caller handle it.

    fp, pathname, description = imp.find_module(name)

    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        # Since we may exit via an exception, close fp explicitly.
        if fp:
            fp.close()
```

CAPÍTULO 38

Módulos Não Documentados

Aqui está uma lista rápida de módulos que não estão documentados no momento, mas que devem ser documentados. Sinta-se à vontade para contribuir com documentação para eles! (Envie por e-mail para docs@python.org.)

A ideia e o conteúdo original deste capítulo foram retirados de uma publicação de Fredrik Lundh; o conteúdo específico deste capítulo foi substancialmente revisado.

38.1 Módulos para plataformas específicas

Estes módulos são utilizados para implementar o módulo `os.path` e não estão documentados além desta menção. Há pouca necessidade de documentar isso.

ntpath — Implementação de `os.path` nas plataformas Win32 e Win64.

posixpath — Implementação de `os.path` no POSIX.

APÊNDICE A

Glossário

>>> O prompt padrão do shell interativo do Python. Normalmente visto em exemplos de código que podem ser executados interativamente no interpretador.

. . . O prompt padrão do shell interativo do Python ao inserir o código para um bloco de código recuado, quando dentro de um par de delimitadores correspondentes esquerdo e direito (parênteses, colchetes, chaves ou aspas triplas) ou após especificar um decorador.

2to3 Uma ferramenta que tenta converter código Python 2.x em código Python 3.x tratando a maioria das incompatibilidades que podem se detectadas com análise do código-fonte e navegação na árvore sintática.

O 2to3 está disponível na biblioteca padrão como *lib2to3*; um ponto de entrada é disponibilizado como `Tools/scripts/2to3`. Veja *2to3 - Tradução Automatizada de Código Python 2 para 3*.

classe base abstrata Classes básicas abstratas complementam tipagem pato, fornecendo uma maneira de definir interfaces quando outras técnicas, como *hasattr()*, seriam desajeitadas ou sutilmente erradas (por exemplo, com métodos mágicos). ABCs introduzem subclasses virtuais, que são classes que não herdam de uma classe mas ainda são reconhecidas por *isinstance()* e *issubclass()*; veja a documentação do módulo *abc*. O Python vem com muitas ABCs internas para estruturas de dados (no módulo *collections.abc*), números (no módulo *numbers*), fluxos (no módulo *io*) , localizadores e carregadores de importação (no módulo *importlib.abc*). Você pode criar suas próprias ABCs com o módulo: mod:*abc*.

Anotação Um rótulo associado a uma variável, um atributo de classe ou um parâmetro de função ou valor de retorno, usado por convenção como: term: *type hint*.

Anotações de variáveis locais não podem ser acessadas em tempo de execução, mas anotações de variáveis globais, atributos de classe e funções são armazenadas no atributo especial: attr: *__annotations__* de módulos, classes e funções, respectivamente.

Ver *variable annotation*, *function annotation*, **PEP 484** e **PEP 526**, que descrevem esta funcionalidade

argumento Um valor passado para um *function* (ou method‘) ao chamar a função. Existem dois tipos de argumento:

- *argumento nomeado*: um argumento precedido por um identificador (por exemplo, `nome=`) na chamada de uma função ou passada como um valor em um dicionário precedido por `**`. Por exemplo, `3` e `5` são ambos argumentos nomeados na chamada da função *complex()* a seguir:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *argumento posicional*: um argumento que não é um argumento nomeado. Argumentos posicionais podem aparecer no início da lista de argumentos e/ou podem ser passados com elementos de um *iterável* precedido por `*`. Por exemplo, `3` e `5` são ambos argumentos posicionais nas chamadas a seguir:

```
complex(3, 5)
complex(*(3, 5))
```

Argumentos são atribuídos às variáveis locais nomeadas no corpo da função. Veja a seção calls para as regras de atribuição. Sintaticamente, qualquer expressão pode ser usada para representar um argumento; avaliada a expressão, o valor é atribuído à variável local.

Veja também o termo *parâmetro* no glossário, a pergunta the difference between arguments and parameters na FAQ, e **PEP 362**.

gerenciador de contexto assíncrono An object which controls the environment seen in an `async with` statement by defining `__aenter__()` and `__aexit__()` methods. Introduced by **PEP 492**.

gerador assíncrono A function which returns an *asynchronous generator iterator*. It looks like a coroutine function defined with `async def` except that it contains `yield` expressions for producing a series of values usable in an `async for` loop.

Normalmente se refere a uma função geradora assíncrona, mas pode se referir a um iterador gerador assíncrono em alguns contextos. Em casos em que o significado não esteja claro, usar o termo completo evita a ambiguidade.

Uma função geradora assíncrona pode conter expressões `await` e também `async for` e `async with`.

gerador iterador assíncrono Um objeto criado por uma função *asynchronous generator*.

Este é um *iterador assíncrono* que, quando chamado usando o método `__anext__()`, retorna um objeto aguardável que executará o corpo da função de gerador assíncrono até a próxima expressão `yield`.

Cada `yield` suspende temporariamente o processamento, lembrando o estado de execução do local (incluindo variáveis locais e instruções de tentativa pendentes). Quando o *iterador do gerador assíncrono* efetivamente é retomado com outro retorno esperado por `__anext__()`, ele inicia de onde parou. Veja **PEP 492** e **PEP 525**.

iterável assíncrono Um objeto que pode ser usado em uma instrução `async for`. Deve retornar um *iterador assíncrono* do seu método `__aiter__()`. Introduzido por **PEP 492**.

iterador assíncrono Um objeto que implementa os métodos `__aiter__()` e `__anext__()`. `__anext__` deve retornar um objeto *aguardável*. `async for` resolve os aguardáveis retornados por um método `__anext__()` do iterador assíncrono até que ele levante uma exceção *`StopAsyncIteration`*. Introduzido pela **PEP 492**.

atributo Um valor associado a um objeto que é referenciado pelo nome separado por um ponto. Por exemplo, se um objeto *o* tem um atributo *a* esse seria referenciado como *o.a*.

aguardável Um objeto que pode ser usado em uma expressão `await`. Pode ser uma *coroutine* ou um objeto com um método `__await__()`. Veja também a **PEP 492**.

BDFL Abreviação da expressão da língua inglesa "Benevolent Dictator for Life" (em português, "Ditador Benevolente Vitalício"), referindo-se a Guido van Rossum, criador do Python.

arquivo binário Um *objeto arquivo* capaz de ler e gravar em *objetos byte ou similar*. Exemplos de arquivos binários são arquivos abertos no modo binário (`'rb'`, `'wb'` ou `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer` e instâncias de *`io.BytesIO`* e *`gzip.GzipFile`*.

Veja também *arquivo texto* para um arquivo objeto capaz de ler e gravar em objetos *`str`*.

objeto byte ou similar Um objeto que suporta o bufferobjects e pode exportar um buffer C *contíguo*. Isso inclui todos os objetos *`bytes`*, *`bytearray`* e *`array.array`*, além de muitos objetos comuns *`memoryview`*. Objetos byte

ou similar podem ser usados para várias operações que funcionam com dados binários; isso inclui compactação, salvamento em um arquivo binário e envio por um soquete.

Algumas operações precisam que os dados binários sejam mutáveis. A documentação geralmente se refere a eles como "objetos byte ou similiar para leitura-escrita". Exemplos de objetos de buffer mutável incluem *bytearray* e um *memoryview* de um *bytearray*. Outras operações exigem que os dados binários sejam armazenados em objetos imutáveis ("objetos byte ou similar para somente leitura"); exemplos disso incluem *bytes* e a *memoryview* de um objeto *bytes*.

bytecode Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This "intermediate language" is said to run on a *virtual machine* that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

A list of bytecode instructions can be found in the documentation for *the dis module*.

Classe A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

variável de classe Uma variável definida em uma classe e destinada a ser modificada apenas no nível da classe (ou seja, não em uma instância da classe).

Coerção A conversão implícita de uma instância de um tipo para outro durante uma operação que envolve dois argumentos do mesmo tipo. Por exemplo, `int(3.15)` converte o número do ponto flutuante no número inteiro `3`, mas em `3+4.5`, cada argumento é de um tipo diferente (um int, um float), e ambos devem ser convertidos para o mesmo tipo antes de poderem ser adicionados ou isso levantará um *TypeError*. Sem coerção, todos os argumentos de tipos compatíveis teriam que ser normalizados com o mesmo valor pelo programador, por exemplo, `float(3)+4.5` em vez de apenas `3+4.5`.

número complexo An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of -1), often written `i` in mathematics or `j` in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a `j` suffix, e.g., `3+1j`. To get access to complex equivalents of the *math* module, use *cmath*. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

gerenciador de contexto An object which controls the environment seen in a `with` statement by defining `__enter__()` and `__exit__()` methods. See **PEP 343**.

variável de contexto Uma variável que pode ter valores diferentes, dependendo do seu contexto. Isso é semelhante ao armazenamento local do encadeamento, no qual cada encadeamento de execução pode ter um valor diferente para uma variável. No entanto, com variáveis de contexto, pode haver vários contextos em um encadeamento de execução e o principal uso para variáveis de contexto é acompanhar as variáveis em tarefas assíncronas simultâneas. Veja *contextvars*.

contíguo Um buffer é considerado contíguo exatamente se for *contíguo C * ou *contíguo Fortran *. Os buffers de dimensão zero são contíguos C e Fortran. Em matrizes unidimensionais, os itens devem ser dispostos na memória próximos um do outro, em ordem crescente de índices, começando do zero. Em matrizes multidimensionais contíguas C, o último índice varia mais rapidamente ao visitar itens em ordem de endereço de memória. No entanto, nas matrizes contíguas do Fortran, o primeiro índice varia mais rapidamente.

co-rotina Coroutines são uma forma mais generalizada de sub-rotinas. Sub-routinas tem a entrada iniciada em um ponto, e a saída em outro ponto. Coroutines podem entrar, sair, e continuar em muitos pontos diferentes. Elas podem ser implementadas com a instrução `async def`. Veja também **PEP 492**.

função de co-rotina Uma função que retorna um objeto do tipo *coroutine*. Uma função coroutine pode ser definida com a instrução `async def`, e pode conter as palavras chaves `await`, `async for`, e `async with`. Isso foi introduzido pela **PEP 492**.

CPython The canonical implementation of the Python programming language, as distributed on python.org. The term "CPython" is used when necessary to distinguish this implementation from others such as Jython or IronPython.

decorador Uma função que retorna outra função, geralmente aplicada como uma transformação de função usando a sintaxe `@wrapper`. Exemplos comuns para decoradores são *`classmethod()`* e *`staticmethod()`*.

A sintaxe do decorador é meramente um açúcar-sintático, as duas definições de funções a seguir são semanticamente equivalentes:

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

O mesmo conceito existe para as classes, mas não é comumente utilizado. Veja a documentação de function definitions e class definitions para obter mais informações sobre decoradores.

descritor Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

Para obter mais informações sobre os métodos dos descritores, veja: descriptors.

dicionário Um Array associativo em que chaves arbitrárias são mapeadas para valores. As chaves podem ser quaisquer objetos que possuam os métodos `__hash__()` e `__eq__()`. Dicionários são estruturas chamadas de hash na linguagem Perl.

visualização de dicionário Os objetos retornados por *`dict.keys()`*, *`dict.values()`* e *`dict.items()`* são chamados de Views de Dicionário. Eles fornecem uma visualização dinâmica das entradas do dicionário, o que significa que quando o dicionário é alterado, a View reflete essas alterações. Para forçar a View do dicionário a se tornar uma lista completa use `list(dictview)`. Veja: ref:*dict-views*.

docstring Uma string literal que aparece como primeira expressão numa classe, função ou módulo. Ainda que sejam ignoradas quando a suíte é executada, é reconhecida pelo compilador que a coloca no atributo `__doc__` da classe, função ou módulo que a encapsula. Como ficam disponíveis por meio de introspecção, docstrings são o lugar canônico para documentação do objeto.

duck-typing (tipagem pato) A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using *`type()`* or *`isinstance()`*. (Note, however, that duck-typing can be complemented with *abstract base classes*.) Instead, it typically employs *`hasattr()`* tests or *EAFP* programming.

EAFP Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the *LBYL* style common to many other languages such as C.

expressão Uma parte da sintaxe que pode ser avaliada para algum valor. Em outras palavras, uma expressão é a acumulação de elementos de expressão como literais, nomes, atributos de acesso, operadores ou chamadas de funções, todos os quais retornam um valor. Em contraste com muitas outras linguagens, nem todas as contruções de linguagem são expressões. Também existem *statement*, os quais não podem ser usadas como expressões, como por exemplo `while`. Atribuições também são instruções, não expressões.

módulo de extensão Um módulo escrito em C ou C++, usando a API C de Python para interagir tanto com código de usuário quanto do núcleo.

f-string Literais string prefixadas com `'f'` ou `'F'` são conhecidas como "f-strings" que é uma abreviação de formatted string literals. Veja também **PEP 498**.

file object (arquivo objeto) Um objeto que expõe uma API orientada a arquivos (com métodos tais como `read()` ou `write()`) para um recurso subjacente. Dependendo da maneira como foi criado, um objeto arquivo pode mediar o acesso a um arquivo real no disco ou outro tipo de dispositivo de armazenamento ou de comunicação (por exemplo a entrada/saída padrão, buffers em memória, sockets, pipes, etc.). Objetos arquivo também são chamados de *file-like objects* ou *streams*.

Atualmente há três categorias de objetos arquivo: arquivos binários raw, .. XXX: sugestões para "raw" e "bufferizados"? arquivos binários bufferizados e arquivos texto. Suas interfaces estão definidas no módulo `io`. A forma canônica de se criar um objeto arquivo é por meio da função `open()`.

file-like object (objeto como a um arquivo) Um sinônimo do termo *file object*.

finder An object that tries to find the *loader* for a module that is being imported.

Desde o Python 3.3, existem dois tipos de localizadores: *meta path finders* para uso com `sys.meta_path`, e *path entry finders* para uso com `sys.path_hooks`.

Veja **PEP 302**, **PEP 420** e **PEP 451** para maiores informações.

divisão pelo piso Mathematical division that rounds down to nearest integer. The floor division operator is `//`. For example, the expression `11 // 4` evaluates to `2` in contrast to the `2.75` returned by float true division. Note that `(-11) // 4` is `-3` because that is `-2.75` rounded *downward*. See **PEP 238**.

function (função) A series of statements which returns some value to a caller. It can also be passed zero or more *arguments* which may be used in the execution of the body. See also *parameter*, *method*, and the function section.

function annotation (anotação de função) Uma *annotation* de um parâmetro ou retorno de uma função.

Anotações de função são comumente usados por *type hints*: por exemplo, essa função espera receber dois argumentos `int` e também é esperado que devolva um valor `int`:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

A sintaxe de anotação de uma função é explicada na seção function.

Veja *variable annotation* e **PEP 484**, que descrevem essa funcionalidade.

__future__ A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter.

Ao importar o módulo `__future__` e avaliar suas variáveis, você pode ver quando uma nova funcionalidade foi adicionada pela primeira vez à linguagem e quando ela se tornará padrão:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection (coletor de lixo) O processo de liberar a memória quando ela não é mais utilizada. Python executa a liberação da memória através da contagem de referências e um coletor de lixo cíclico que é capaz de detectar e interromper referências cíclicas. O coletor de lixo pode ser controlado usando o módulo `gc`.

gerador A function which returns a *generator iterator*. It looks like a normal function except that it contains `yield` expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function.

Normalmente refere-se a uma função geradora, mas pode referir-se a um *iterador gerador* em alguns contextos. Em alguns casos onde o significado desejado não está claro, usar o termo completo evita ambiguidade.

iterador gerador Um objeto criado por uma função *gerador*.

Cada `yield` suspende temporariamente o processamento, memorizando o estado da execução local (incluindo variáveis locais e instruções try pendentes). Quando o *iterador gerador* retorna, ele se recupera do último ponto onde estava (em contrapartida as funções que iniciam uma nova execução a cada vez que são invocadas).

generator expression Uma expressão que retorna um iterador. Parece uma expressão normal, seguido de uma cláusula `for` definindo uma variável de loop, um range, e uma cláusula `if` opcional. A expressão combinada gera valores para uma função encapsuladora:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

generic function (função genérica) Uma função composta por multiplas funções implementando a mesma operação para diferentes tipos. Qual implementação deverá ser usada durante a execução é determinada pelo algoritmo de despacho.

Veja também a entrada *single dispatch* no glossário, o decorador *`functools.singledispatch()`*, e a **PEP 443**.

GIL Veja *global interpreter lock*.

global interpreter lock (bloqueio global do intérprete) The mechanism used by the *CPython* interpreter to assure that only one thread executes Python *bytecode* at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as *`dict`*) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

Past efforts to create a "free-threaded" interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case. It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.

pyc baseado em hash Um arquivo de cache em bytecode que usa hash ao invés do tempo (no qual o arquivo de código-fonte foi modificado pela última vez) para determinar a sua validade. Veja pyc-invalidation.

hashable An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

A maioria dos objetos embutidos imutáveis do Python são hasheáveis; containers mutáveis (tais como listas ou dicionários) não são; containers imutáveis (tais como tuplas e frozensets) são hasheáveis apenas se os seus elementos são hasheáveis. Objetos que são instâncias de classes definidas pelo usuário são hasheáveis por padrão. Todos eles comparam de forma desigual (exceto entre si mesmos), e o seu valor hash é derivado a partir do seu *`id()`*.

IDLE Um ambiente de desenvolvimento integrado para Python. IDLE é um editor básico e um ambiente interpretador que vem junto com a distribuição padrão do Python.

imutável Um objeto que possui um valor fixo. Objetos imutáveis incluem números, strings e tuplas. Estes objetos não podem ser alterados. Um novo objeto deve ser criado se um valor diferente tiver de ser armazenado. Objetos imutáveis têm um papel importante em lugares onde um valor constante de hash seja necessário, como por exemplo uma chave em um dicionário.

import path Uma lista de localizações (ou *path entries*) que são buscadas pelo *path based finder* por módulos para importar. Durante a importação, esta lista de localizações usualmente vem a partir de `sys.path`, mas para sub-pacotes ela também pode vir do atributo `__path__` de pacotes-pai.

importando O processo pelo qual o código Python em um módulo é disponibilizado para o código Python em outro módulo.

importer Um objeto que localiza e carrega um módulo; Tanto um *finder* e o objeto *loader*.

interactive Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`).

interpretado Python é uma linguagem interpretada, em oposição àquelas que são compiladas, embora esta distinção possa ser nebulosa devido à presença do compilador de bytecode. Isto significa que os arquivos-fontes podem ser executados diretamente sem necessidade explícita de se criar um arquivo executável. Linguagens interpretadas normalmente têm um ciclo de desenvolvimento/depuração mais curto que as linguagens compiladas, apesar de seus programas geralmente serem executados mais lentamente. Veja também interativo.

interpreter shutdown Quando solicitado para desligar, o interpretador Python entra em uma fase especial, onde ele gradualmente libera todos os recursos alocados, tais como módulos e várias estruturas internas críticas. Ele também faz diversas chamadas para o *garbage collector*. Isto pode disparar a execução de código em destrutores definidos pelo usuário ou callbacks de referência fraca. Código executado durante a fase de desligamento pode encontrar diversas exceções, pois os recursos que ele depende podem não funcionar mais (exemplos comuns são os módulos de bibliotecas, ou os mecanismos de avisos).

A principal razão para o interpretador desligar, é que o módulo `__main__` ou o script sendo executado terminou sua execução.

iterável Um objeto capaz de retornar seus membros um de cada vez. Exemplos de iteráveis incluem todos os tipos de sequência (tais como `list`, `str` e `tuple`) e alguns tipos de não-sequência, como o `dict`, *file objects*, além dos objetos de quaisquer classes que você definir com um método `__iter__()` ou `__getitem__()` que implementam a semântica de *sequência* .

Iteráveis podem ser usados em um laço `for` e em vários outros lugares em que uma sequência é necessária (`zip()`, `map()`, ...). Quando um objeto iterável é passado como argumento para a função nativa `iter()`, ela retorna um iterador para o objeto. Este iterador é adequado para se varrer todo o conjunto de valores. Ao usar iteráveis, normalmente não é necessário chamar `iter()` ou lidar com os objetos iteradores em si. A instrução `for` faz isso automaticamente para você, criando uma variável temporária para armazenar o iterador durante a execução do laço. Veja também *iterador*, *sequência*, e *gerador*.

iterador Um objeto que representa um fluxo de dados. Repetidas chamadas ao método `__next__()` de um iterador (ou passando o objeto para a função nativa `next()`) vão retornar itens sucessivos do fluxo. Quando não houver mais dados disponíveis uma exceção `StopIteration` exception será levantada. Neste ponto, o objeto iterador se esgotou e quaisquer chamadas subsequentes a seu método `__next__()` vão apenas levantar a exceção `StopIteration` novamente. Iteradores precisam ter um método `__iter__()` que retorne o objeto iterador em si, de forma que todo iterador também é iterável e pode ser usado na maioria dos lugares em que um iterável é requerido. Uma notável exceção é código que tenta realizar passagens em múltiplas iterações. Um objeto contêiner (como uma `list`) produz um novo iterador a cada vez que você passá-lo para a função `iter()` ou utilizá-lo em um laço `for`. Tentar isso com o mesmo iterador apenas iria retornar o mesmo objeto iterador esgotado já utilizado na iteração anterior, como se fosse um contêiner vazio.

Mais informações podem ser encontradas em *Tipos de Iteradores*.

Função chave A key function or collation function is a callable that returns a value used for sorting or ordering. For example, `locale.strxfrm()` is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.`

nlargest(), and *itertools.groupby()*.

There are several ways to create a key function. For example. the *str.lower()* method can serve as a key function for case insensitive sorts. Alternatively, a key function can be built from a `lambda` expression such as `lambda r: (r[0], r[2])`. Also, the *operator* module provides three key function constructors: *attrgetter()*, *itemgetter()*, and *methodcaller()*. See the Sorting HOW TO for examples of how to create and use key functions.

keyword argument (Argumento de Palavra-Chave) Veja o *argument*.

lambda Uma função de linha anônima consistindo de uma única *expression*, que é avaliada quando a função é chamada. A sintaxe para criar uma função lambda é `lambda [parameters]: expression`

LBYL Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the *EAFP* approach and is characterized by the presence of many `if` statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between "the looking" and "the leaping". For example, the code, `if key in mapping: return mapping[key]` can fail if another thread removes *key* from *mapping* after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

list Uma *sequence* embutida no Python. Apesar do seu nome, é mais próximo de um vetor em outras linguagens do que uma lista encadeada, como o acesso aos elementos é da ordem O(1).

list comprehension A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

carregador An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See **PEP 302** for details and *importlib.abc.Loader* for an *abstract base class*.

método mágico Um sinônimo informal para um *special method*.

mapeando A container object that supports arbitrary key lookups and implements the methods specified in the *Mapping* or *MutableMapping* *abstract base classes*. Examples include *dict*, *collections.defaultdict*, *collections.OrderedDict* and *collections.Counter*.

meta path finder Um *finder* retornado por uma busca de *sys.meta_path*. Meta localizadores de diretórios são relacionados a, mas diferentes de *path entry finders*.

Veja *importlib.abc.MetaPathFinder* para os métodos que meta localizadores de diretórios implementam.

metaclass The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in metaclasses.

method (método) A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first *argument* (which is usually called `self`). See *function* and *nested scope*.

method resolution order (ordem de resolução de método) Method Resolution Order is the order in which base classes are searched for a member during lookup. See The Python 2.3 Method Resolution Order for details of the algorithm used by the Python interpreter since the 2.3 release.

módulo Um objeto que serve como uma unidade organizacional de código Python. Os módulos têm um namespace contendo objetos Python arbitrários. Os módulos são carregados pelo Python através do processo de: term: *importing*.

Veja também *package*.

module spec (módulo spec) Uma namespace que contém as informações relacionadas à importação usadas para carregar um módulo. Uma instância de class:*importlib.machinery.ModuleSpec*.

MRO See *method resolution order*.

mutable (mutável) Objeto mutável é aquele que pode modificar seus valor mas manter seu `id()`. Veja também *immutable*.

named tuple O termo "tupla nomeada" é aplicado a qualquer tipo ou classe que herda de tupla e cujos elementos indexáveis também são acessíveis usando atributos nomeados. O tipo ou classe pode ter outras funcionalidades também.

Diversos tipos embutidos são tuplas nomeadas, incluindo os valores retornados por `time.localtime()` e `os.stat()`. Outro exemplo é `sys.float_info`:

```
>>> sys.float_info[1]                    # indexed access
1024
>>> sys.float_info.max_exp               # named field access
1024
>>> isinstance(sys.float_info, tuple)    # kind of tuple
True
```

Algumas tuplas nomeadas são tipos embutidos (tal como os exemplos acima). Alternativamente, uma tupla nomeada pode ser criada a partir da definição de uma classe regular, que herde de `tuple` e que defina campos nomeados. Tal classe pode ser escrita a mão, ou ela pode ser criada com uma função factory `collections.namedtuple()`. A segunda técnica também adiciona alguns métodos extras, que podem não ser encontrados quando foi escrita manualmente, ou em tuplas nomeadas embutidas.

namespace O lugar em que uma variável é armazenada. Namespaces são implementados como dicionários. Existem os namespaces local, global e nativo, bem como namespaces aninhados em objetos (em métodos). Namespaces suportam modularidade ao previnir conflitos de nomes. Por exemplo, as funções `__builtin__.open()` e `os.open()` são diferenciadas por seus namespaces. Namespaces também auxiliam na legibilidade e na manutenibilidade ao torar mais claro quais módulos implementam uma função. Escrever `random.seed()` ou `itertools.izip()`, por exemplo, deixa claro que estas funções são implementadas pelos módulos `random` e `itertools` respectivamente.

namespace package (espaço de nomes do pacote) Um *package* **PEP 420** que serve apenas como container para sub pacotes. Pacotes de namespaces podem não ter representação física, e especificamente não são como um *regular package* porque eles não tem um arquivo `__init__.py`.

Veja também *module*.

nested scope (escopo aninhado) A habilidade de referir-se a uma variável em uma definição de fechamento. Por exemplo, uma função definida dentro de outra pode referenciar variáveis da função externa. Perceba que escopos aninhados por padrão funcionam apenas por referência e não por atribuição. Variáveis locais podem ler e escrever no escopo mais interno. De forma similar, variáveis globais podem ler e escrever para o namespace global. O `nonlocal` permite escrita para escopos externos.

new-style class (novo estilo de classes) Antigo nome para o tipo de classes agora usado para todos os objetos de classes. Em versões anteriores do Python, apenas classes com o novo estilo podiam usar recursos novos e versáteis do Python, tais como `__slots__`, descritores, propriedades, `__getattribute__()`, métodos de classe, e métodos estáticos.

object (objeto) Qualquer dado que tenha estado (atributos ou valores) e comportamento definidos (métodos). Também a última classe base de qualquer *new-style class*.

pacote Um *module* Python é capaz de conter submódulos ou recursivamente, sub-pacotes. Tecnicamente, um pacote é um módulo Python com um atributo `__path__`.

Veja também *regular package* e *namespace package*.

parameter (parâmetro) Uma entidade nomeada na definição de uma *função* (ou método) que específica um *argumento* (ou em alguns casos, argumentos) que a função pode receber. Existem cinco tipos de parâmetros:

- *posicional-ou-nomeado*: especifica um argumento que pode ser tanto *posicional* quanto *nomeado*. Esse é o tipo padrão de parâmetro, por exemplo *foo* e *bar* a seguir:

```
def func(foo, bar=None): ...
```

- *somente-posicional*: especifica um argumento que pode ser passado para a função somente por posição. Python não possui sintaxe para definir parâmetros somente-posicionais. Contudo, algumas funções embutidas possuem argumentos somente-posicionais (por exemplo, `abs()`).

- *somente-nomeado*: especifica um argumento que pode ser passado para a função somente por nome. Parâmetros somente-nomeados podem ser definidos com um simples parâmetro var-posicional ou um `*` antes deles na lista de parâmetros na definição da função, por exemplo *kw_only1* and *kw_only2* a seguir:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-posicional*: especifica quem uma sequência arbitrária de argumentos posicionais pode ser fornecida (em adição a qualquer argumento posicional já aceito por outros parâmetros). Tal parâmetro pode ser definido colocando um `*` antes do nome, por exemplo *args* a seguir:

```
def func(*args, **kwargs): ...
```

- *var-nomeado*: especifica que, arbitrariamente, muitos argumentos nomeados podem ser fornecidos (em adição a qualquer argumento nomeado já aceito por outros parâmetros). Tal parâmetro pode definido colocando-se `**` antes do nome, por exemplo *kwargs* no exemplo acima.

Parâmetros podem especificar tanto argumentos opcionais quanto obrigatórios, assim como valores padrões para alguns argumentos opcionais.

Veja o termo *argument* no glossario, a questão :ref:‘ sobre a diferença entre argumentos e parâmetros <faq-argument-vs-parameter>‘ na FAQ, a classe `inspect.Parameter`, a seçã função, e **PEP 362**.

entrada de caminho Um local único no term:*import path* que o *path based finder* consulta para encontrar módulos a serem importados.

path entry finder (localizador de entrada de path) Um *finder* retornado por um callable em `sys.path_hooks` (ou seja, um *path entry hook*) que sabe como localizar os módulos *path entry*.

Veja `importlib.abc.PathEntryFinder` para os métodos implementadores da entrada do path.

path entry hook (hook do path de entrada) Um callable na lista `sys.path_hook` que retorna um *path entry finder* caso saiba como encontrar módulos em um local específico *path entry*.

path based finder Uma das opções padrãot *meta path finders* que será procurado por módulos *import path*.

objeto caminho ou similar Um objeto representando um arquivo de caminho do sistema. Um objeto caminho ou similar é ou um objeto `str` ou `bytes` representando um caminho, ou um objeto implementando o protocolo `os.PathLike`. Um objeto que suporta o protocolo `os.PathLike` pode ser convertido para um arquivo de caminho do sistema `str` ou `bytes`, através da chamada da função `os.fspath()`; `os.fsdecode()` e `os.fsencode()` podem ser usadas para garantir um `str` ou `bytes` como resultado, respectivamente. Introduzido na **PEP 519**.

PEP Proposta de melhoria do Python. Uma PEP é um documento de design que fornece informação para a comunidade Python, ou descreve uma nova funcionalidade para o Python ou seus predecessores ou ambientes. PEPs devem prover uma especificação técnica concisa e um racional para funcionalidades propostas.

PEPs tem a intenção de ser os mecanismos primários para propor novas funcionalidades significativas, para coletar opiniões da comunidade sobre um problema, e para documentar as decisões de design que foram adicionadas ao Python. O autor da PEP é responsável por construir um consenso dentro da comunidade e documentar opiniões dissidentes.

Veja **PEP 1**.

parte Um conjunto de arquivos em um único diretório (possivelmente armazenado em um arquivo zip) que contribuem para um pacote de namespace, conforme definido em **PEP 420**.

positional argument (argumento posicional) Veja o *argument*.

API provisória Uma API provisória é uma API que foi deliberadamente excluída das bibliotecas padrões com compatibilidade retroativa garantida. Enquanto mudanças maiores para tais interfaces não são esperadas, contanto que elas sejam marcadas como provisórias, mudanças retroativas incompatíveis (até e incluindo a remoção da interface) podem ocorrer se consideradas necessárias pelos desenvolvedores principais. Tais mudanças não serão feitas gratuitamente – elas irão ocorrer apenas se sérias falhas fundamentais forem descobertas, que foram esquecidas anteriormente a inclusão da API.

Mesmo para APIs provisórias, mudanças retroativas incompatíveis são vistas como uma "solução em último caso" - cada tentativa ainda será feita para encontrar uma resolução retroativa compatível para quaisquer problemas encontrados.

Esse processo permite que a biblioteca padrão continue a evoluir com o passar do tempo, sem se prender em erros de design problemáticos por períodos de tempo prolongados. Veja **PEP 411** para mais detalhes.

pacote provisório Veja *provisional API*.

Python 3000 Apelido para a versão do Python 3.x linha de lançamento (cunhado há muito tempo, quando o lançamento da versão 3 era algo em um futuro muito distante.) Esse termo possui a seguinte abreviação: "Py3k".

Pythonic Uma ideia ou um pedaço de código que segue de perto os idiomas mais comuns da linguagem Python, ao invés de implementar códigos usando conceitos comuns a outros idiomas. Por exemplo, um idioma comum em Python é fazer um loop sobre todos os elementos de uma iterável usando a instrução: *for* statement. Muitas outras línguas não têm esse tipo de construção, então as pessoas que não estão familiarizadas com o Python usam um contador numérico:

```
for i in range(len(food)):
    print(food[i])
```

Ao contrário do método limpo, ou então, Pythônico:

```
for piece in food:
    print(piece)
```

qualified name (nome qualificado) Um nome pontilhado (quando 2 termos são ligados por um ponto) que mostra o "path" do escopo global de um módulo para uma classe, função ou método definido num determinado módulo, conforme definido pela **PEP 3155**. Para funções e classes de nível superior, o nome qualificado é o mesmo que o nome do objeto:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
```

(continua na próxima página)

(continuação da página anterior)

```
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Quando usado para se referir a módulos, o *fully qualified name* significa todo o caminho pontilhado para o módulo, incluindo quaisquer pacotes pai, por exemplo: `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count O número de refrências para um objeto. Quando a contagem de referências de um objeto atinge zero, ele é desalocado. Contagem de referências geralmente não é visível no código Python, mas é um elemento chave da implementação *CPython*. O módulo `sys` define a função `getrefcount()` que programadores podem chamar para retornar a contagem de referências para um objeto em particular.

regular package Um *package* tradicional, como um diretório contendo um arquivo `__init__.py`.

Veja também *namespace package*.

__slots__ A declaração dentro de uma classe que salva memória através de pré-declarações de espaço para atributos des instâncias, e eliminando dicionários de instâncias. Apesar de popular, a técnica é um tanto quanto complicada de acertar, e é melhor se for reservada para casos raros, onde existe uma grande quantidade de instâncias em uma aplicação onde a memória é crítica.

sequência Um *iterable* com suporte para acesso eficiente a seus elementos através de índices inteiros via método especial `__getitem__()` e que define o método `__len__()` que devolve o tamanho da sequência. Alguns tipos de sequência nativos são: `list`, `str`, `tuple`, e `bytes`. Note que `dict` também tem suporte para `__getitem__()` e `__len__()`, mas é considerado um mapa e não uma sequência porque a busca usa uma chave *imutável* arbitrária em vez de inteiros.

A classe base abstrata `collections.abc.Sequence` define uma interface mais rica que vai além de apenas `__getitem__()` e `__len__()`, adicionando `count()`, `index()`, `__contains__()`, e `__reversed__()`. Tipos que implementam essa interface podem ser explicitamente registrados usando `register()`.

single dispatch (despacho único) Uma forma do *generic function* despacho onde a implementação é escolhida com base no tipo de um único argumento.

slice Um objeto geralmente contendo uma parte de uma *sequence*. Uma fatia é criada usando a notação de subscrito `[]` pode conter também até dois pontos entre números, como em `variable_name[1:3:5]`. A notação de suporte (subscrito) utiliza objetos `slice` internamente.

método especial Um método que é chamado implicitamente pelo Python para executar uma certa operação em um tipo, como uma adição por exemplo. Tais métodos tem nomes iniciando e terminando com dois underscores. Métodos especiais sestão documentados em specialnames.

declaração Uma instrução é parte de uma suíte (um "bloco" de código). Uma instrução é ou uma *expression* ou uma de várias construções com uma palavra-chave, tal como `if`, `while` ou `for`.

codificador de texto Um codec que codifica strings Unicode para bytes.

arquivo texto Um *file object* apto a ler e escrever objetos `str`. Geralmente, um arquivo texto, na verdade, acesse um fluxo de dados de bytes e captura o *text encoding* automaticamente. Exemplos de arquivos texto são: arquivos abertos em modo texto (`'r'` or `'w'`), `sys.stdin`, `sys.stdout`, e instâncias de `io.StringIO`.

Veja também *binary file* para um objeto arquivo apto a ler e escrever *bytes-like objects*.

aspas triplas Uma string que está definida com três ocorrências de aspas duplas (") ou apóstrofes ('). Enquanto elas não fornecem nenhuma funcionalidade não disponível com strings de aspas simples, elas são úteis para inúmeras razões. Elas permitem que você inclua aspas simples e duplas não encerradas dentro de uma string, e elas podem utilizar múltiplas linhas sem o uso de caracteres de continuação, fazendo-as especialmente úteis quando escrevemos documentação em docstrings.

tipo O tipo de um objeto Python determina qual tipo de objeto ele é; todos objetos tem um tipo. Um tipo de objeto é acessivel pelo atributo *__class__* ou pode ser recuperado com `type(obj)`.

tipo alias Um sinônimo para tipo, criado através da atribuição do tipo para um identificador.

Tipos alias são úteis para simplificar *type hints*. Por exemplo:

```
from typing import List, Tuple

def remove_gray_shades(
        colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

pode tornar-se mais legível desta forma:

```
from typing import List, Tuple

Color = Tuple[int, int, int]

def remove_gray_shades(colors: List[Color]) -> List[Color]:
    pass
```

Veja *typing* e **PEP 484**, o qual descreve esta funcionalidade.

dica do tipo Uma *annotation* que especifica o tipo esperado para uma variável, um atributo de classe, ou um parâmetro de função ou um valor de retorno.

Dicas de tipo são opcionais e não são forçadas pelo Python, mas elas são úteis para ferramentas de análise estática de tipos, e ajudam IDEs a completar e refatorar código.

Dicas de tipos de variáveis globais, atributos de classes, e funções, mas não de variáveis locais, podem ser acessadas usando *typing.get_type_hints()*.

Veja *typing* e **PEP 484**, o qual descreve esta funcionalidade.

Novas linhas universais Uma maneira de interpretar fluxos de textos, na qual todos estes são reconhecidos como caracteres de encerramento de linha: a convenção para fim-de-linha no Unix `'\n'`, a convenção no Windows `'\r\n'`, e a antiga convenção no Macintosh `'\r'`. Veja **PEP 278** e **PEP 3116**, bem como *bytes.splitlines()* para uso adicional.

anotação variável Uma *annotation* de uma variável ou um atributo de classe.

Ao fazer uma anotação de uma variável ou atributo de classe, a atribuição é opcional:

```
class C:
    field: 'annotation'
```

Variable annotations are usually used for *type hints*: for example this variable is expected to take *int* values:

```
count: int = 0
```

A sintaxe de anotação de variável é explicada na seção annassign.

Veja *function annotation*, **PEP 484** e **PEP 526**, que descrevem esta funcionalidade.

ambiente virtual Um ambiente de execução isolado que permite usuários Python e aplicações instalarem e atualizarem pacotes Python sem interferir no comportamento de outras aplicações Python em execução no mesmo sistema.

Veja também *venv*.

virtual machine Um computador definido inteiramente em software. A máquina virtual de Python executa o *bytecode* emitido pelo compilador de bytecode.

Zen of Python Lista de princípios de projeto e filosofias do Python que são úteis para a compreensão e uso da linguagem. A lista é exibida quando se digita "`import this`" no console interativo.

APÊNDICE B

Sobre a Documentação

Estes documentos são gerados a partir de fontes reStructuredText utilizando Sphinx, um processador de documentos escrito especificamente para a documentação do Python.

Desenvolvimento da documentação e suas ferramentas é um esforço totalmente voluntário, como o Python em si. Se você quer contribuir, por favor dê uma olhada na página reporting-bugs para informações sobre como fazer. Novos voluntários são sempre bem vindos!

Agradecimentos especiais para:

- Fred L. Drake, Jr., o criador do primeiro conjunto de ferramentas para documentar o Python e escritor de boa parte do conteúdo;
- O projeto Docutils por ter criado reStructuredText e a suíte Docutils;
- Fredrik Lundh por sua Referência Alternativa para Python projeto do qual, Sphinx tirou muitas idéias boas.

B.1 Contribuidores da Documentação do Python

Muitas pessoas tem contribuído para a linguagem Python, sua biblioteca padrão e sua documentação. Veja Misc/ACKS na distribuição do código-fonte do Python para ver uma lista parcial de contribuidores.

Tudo isso só foi possível com o esforço e a contribuição da comunidade Python, por isso temos essa maravilhosa documentação – Obrigado a todos!

APÊNDICE C

História e Licença

C.1 História do software

O Python foi criado no início dos anos 1990 por Guido van Rossum na Stichting Mathematisch Centrum (CWI, veja https://www.cwi.nl/) na Holanda como um sucessor de uma linguagem chamada ABC. Guido continua a ser o principal autor de Python, embora inclua muitas contribuições de outros.

Em 1995, Guido continuou seu trabalho em Python na Corporação para Iniciativas Nacionais de Pesquisa (CNRI, veja https://www.cnri.reston.va.us/) em Reston, Virgínia, onde lançou várias versões do software.

Em maio de 2000, Guido e a equipe de desenvolvimento principal do Python foram para BeOpen.com para formar a equipe do BeOpen PythonLabs. Em outubro do mesmo ano, a equipe do PythonLabs mudou-se para a Digital Creations (agora Zope Corporation; consulte https://www.zope.org/). Em 2001, a Python Software Foundation (PSF, consulte https://www.python.org/psf/) foi formada, uma organização sem fins lucrativos criada especificamente para possuir a Propriedade Intelectual relacionada ao Python. A Zope Corporation é um membro patrocinador do PSF.

Todas as versões do Python são de código aberto (consulte https://opensource.org/ para a definição de código aberto). Historicamente, a maioria, mas não todas, versões do Python também são compatíveis com GPL; a tabela abaixo resume os vários lançamentos.

Release	Derivado de	Ano	Proprietário	GPL compatível?
0.9.0 a 1.2	n/a	1991-1995	CWI	sim
1.3 a 1.5.2	1.2	1995-1999	CNRI	sim
1.6	1.5.2	2000	CNRI	não
2.0	1.6	2000	BeOpen.com	não
1.6.1	1.6	2001	CNRI	não
2.1	2.0+1.6.1	2001	PSF	não
2.0.1	2.0+1.6.1	2001	PSF	sim
2.1.1	2.1+2.0.1	2001	PSF	sim
2.1.2	2.1.1	2002	PSF	sim
2.1.3	2.1.2	2002	PSF	sim
2.2 e acima	2.1.1	2001-agora	PSF	sim

Nota: Compatível com GPL não significa que estamos distribuindo Python sob a GPL. Todas as licenças do Python, ao contrário da GPL, permitem distribuir uma versão modificada sem fazer alterações em código aberto. As licenças compatíveis com GPL possibilitam combinar o Python com outro software lançado sob a GPL; os outros não.

Graças aos muitos voluntários externos que trabalharam sob a direção de Guido para tornar esses lançamentos possíveis.

C.2 Termos e condições para acessar ou usar Python

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.7.17

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   3.7.17 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 3.7.17 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.7.17 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.7.17 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python
   3.7.17.

4. PSF is making Python 3.7.17 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION␣
↪OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT␣
↪THE
   USE OF PYTHON 3.7.17 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.7.17
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.7.17, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
```

```
6. This License Agreement will automatically terminate upon a material breach␣
↪of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This␣
↪License
   Agreement does not grant permission to use PSF trademarks or trade name in␣
↪a
   trademark sense to endorse or promote products or services of Licensee, or␣
↪any
   third party.

8. By copying, installing or otherwise using Python 3.7.17, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 ACORDO DE LICENÇA DA BEOPEN.COM PARA PYTHON 2.0

CONTRATO DE LICENÇA DE FONTE ABERTA DO BEOPEN PYTHON VERSÃO 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
```

(continua na próxima página)

(continuação da página anterior)

```
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CONTRATO DE LICENÇA DA CNRI PARA O PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   Internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
```

(continua na próxima página)

(continuação da página anterior)

```
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 ACORDO DE LICENÇA DA CWI PARA PYTHON 0.9.0 A 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.3 Licenças e Reconhecimentos para Software Incorporado

Esta seção é uma lista incompleta, mas crescente, de licenças e confirmações para softwares de terceiros incorporados na distribuição do Python.

C.3.1 Mersenne Twister

O módulo: mod: *_random* inclui código baseado em um download de http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. A seguir estão os comentários literais do código original:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).
```

(continua na próxima página)

(continuação da página anterior)

```
Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

O módulo: mod: *socket* usa as funções: func:‘ getaddrinfo‘ e: func: *getnameinfo*, que são codificadas em arquivos de origem separados do Projeto WIDE, http://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.
```

(continua na próxima página)

(continuação da página anterior)

```
THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Serviços de soquete assíncrono

Os módulos: mod: *asynchat* e: mod:‘ asyncore‘ contêm o seguinte aviso

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Gerenciamento de cookies

O módulo: mod: *http.cookies* contém o seguinte aviso

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.
```

(continua na próxima página)

(continuação da página anterior)

```
Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Rastreamento de execução

O módulo: mod: *trace* contém o seguinte aviso

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 Funções UUencode e UUdecode

O módulo: mod: *uu* contém o seguinte aviso

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
```

(continua na próxima página)

(continuação da página anterior)

```
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 Chamadas de Procedimento Remoto XML

O módulo: mod: *xmlrpc.client* contém o seguinte aviso

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

O módulo: mod: *test_epoll* contém o seguinte aviso

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Selecione o kqueue

O módulo: mod: *select* contém o seguinte aviso para a interface do kqueue

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

O arquivo: file: *Python / pyhash.c* contém a implementação de Marek Majkowski do algoritmo SipHash24 de Dan Bernstein. Contém a seguinte nota

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod e dtoa

O arquivo: file: *Python / dtoa.c*, que fornece as funções C dtoa e strtod para conversão de duplas de C para e de strings, é derivado do arquivo com o mesmo nome de David M. Gay, atualmente disponível em http://www.netlib.org/fp/. O arquivo original, conforme recuperado em 16 de março de 2009, contém os seguintes avisos de direitos autorais e de licenciamento

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

Os módulos: mod: *hashlib*,: mod:‘ posix‘,: mod: *ssl*,: mod:‘ crypt‘ usam a biblioteca OpenSSL para desempenho adicional se forem disponibilizados pelo sistema operacional. Além disso, os instaladores do Windows e do Mac OS X para Python podem incluir uma cópia das bibliotecas do OpenSSL, portanto incluímos uma cópia da licença do OpenSSL aqui:

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   *
   * 1. Redistributions of source code must retain the above copyright
   *    notice, this list of conditions and the following disclaimer.
   *
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in
   *    the documentation and/or other materials provided with the
   *    distribution.
   *
   * 3. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   *    endorse or promote products derived from this software without
   *    prior written permission. For written permission, please contact
   *    openssl-core@openssl.org.
   *
   * 5. Products derived from this software may not be called "OpenSSL"
   *    nor may "OpenSSL" appear in their names without prior written
   *    permission of the OpenSSL Project.
   *
   * 6. Redistributions of any form whatsoever must retain the following
   *    acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
   *
   * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
   * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
   * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
   * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
   * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
```

(continua na próxima página)

(continuação da página anterior)

```
    * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
    * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
    * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
    * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
    * OF THE POSSIBILITY OF SUCH DAMAGE.
    * ====================================================================
    *
    * This product includes cryptographic software written by Eric Young
    * (eay@cryptsoft.com).  This product includes software written by Tim
    * Hudson (tjh@cryptsoft.com).
    *
    */

Original SSLeay License
-----------------------

   /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
    * All rights reserved.
    *
    * This package is an SSL implementation written
    * by Eric Young (eay@cryptsoft.com).
    * The implementation was written so as to conform with Netscapes SSL.
    *
    * This library is free for commercial and non-commercial use as long as
    * the following conditions are aheared to.  The following conditions
    * apply to all code found in this distribution, be it the RC4, RSA,
    * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
    * included with this distribution is covered by the same copyright terms
    * except that the holder is Tim Hudson (tjh@cryptsoft.com).
    *
    * Copyright remains Eric Young's, and as such any Copyright notices in
    * the code are not to be removed.
    * If this package is used in a product, Eric Young should be given attribution
    * as the author of the parts of the library used.
    * This can be in the form of a textual message at program startup or
    * in documentation (online or textual) provided with the package.
    *
    * Redistribution and use in source and binary forms, with or without
    * modification, are permitted provided that the following conditions
    * are met:
    * 1. Redistributions of source code must retain the copyright
    *    notice, this list of conditions and the following disclaimer.
    * 2. Redistributions in binary form must reproduce the above copyright
    *    notice, this list of conditions and the following disclaimer in the
    *    documentation and/or other materials provided with the distribution.
    * 3. All advertising materials mentioning features or use of this software
    *    must display the following acknowledgement:
    *    "This product includes cryptographic software written by
    *     Eric Young (eay@cryptsoft.com)"
    *    The word 'cryptographic' can be left out if the rouines from the library
    *    being used are not cryptographic related :-).
    * 4. If you include any Windows specific code (or a derivative thereof) from
    *    the apps directory (application code) you must include an acknowledgement:
    *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
    *
    * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
```

(continua na próxima página)

(continuação da página anterior)

```
    * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
    * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
    * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
    * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
    * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
    * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
    * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
    * SUCH DAMAGE.
    *
    * The licence and distribution terms for any publically available version or
    * derivative of this code cannot be changed.  i.e. this code cannot simply be
    * copied and put under another distribution licence
    * [including the GNU Public Licence.]
    */
```

C.3.13 expat

A extensão: mod: *pyexpat* é construída usando uma cópia incluída das fontes de expatriadas, a menos que a compilação esteja configurada `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

A extensão: mod: *_ctypes* é construída usando uma cópia incluída das fontes libffi, a menos que a compilação esteja configurada `--with-system-libffi`:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

A extensão: mod: *zlib* é construída usando uma cópia incluída das fontes zlib se a versão do zlib encontrada no sistema for muito antiga para ser usada na construção

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

A implementação da tabela de hash usada pelo: mod: *tracemalloc* é baseada no projeto cfuhash

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

O módulo: mod: *_decimal* é construído usando uma cópia incluída da biblioteca libmpdec, a menos que a compilação esteja configurada `--with-system-libmpdec`:

```
Copyright (c) 2008-2016 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
```

(continua na próxima página)

(continuação da página anterior)

```
THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

APÊNDICE D

Direitos Autorais

Veja: *História e Licença* para informações completas de licença e permissões.

Referências Bibliográficas

[Frie09] Friedl, Jeffrey. Mastering Regular Expressions. 3ª ed., O'Reilly Media, 2009. A terceira edição do livro não mais cobre Python, mas a primeira edição cobriu a escrita de bons padrões de expressão regular em grandes detalhes.

[C99] ISO/IEC 9899:1999. "Programming languages – C." A public draft of this standard is available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

Índice de Módulos Python

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

W

X

Z

Não alfabético

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z