The Python Language Reference

Release 3.6.15

Guido van Rossum
and the Python development team

setembro 06, 2021

Python Software Foundation
Email: docs@python.org

Sumário

Este manual de referência descreve a sintaxe e a "semântica central" da linguagem. É conciso, mas tenta ser exato e completo. A semântica dos tipos de objetos internos não essenciais e das funções e módulos internos é descrita em library-index. Para uma introdução informal à linguagem, consulte tutorial-index. Para programadores em C ou C++, existem dois manuais adicionais: extending-index descreve a imagem de alto nível de como escrever um módulo de extensão Python, e o c-api-index descreve as interfaces disponíveis para programadores C/C++ em detalhes.

CAPÍTULO 1

Introdução

Este manual de referência descreve a linguagem de programação Python. O mesmo não tem como objetivo de ser um tutorial.

Enquanto estou tentando ser o mais preciso possível, optei por usar especificações em inglês e não formal para tudo, exceto para a sintaxe e análise léxica. Isso deve tornar o documento mais compreensível para o leitor intermediário, mas deixará margem para ambiguidades. Consequentemente, caso estivesses vindo de Marte e tentasse re-implementar o Python a partir deste documento, sozinho, talvez precisarias adivinhar algumas coisas e, na verdade, provavelmente acabaria por implementar um linguagem bem diferente. Por outro lado, se estiveres usando o Python e se perguntando quais são as regras precisas sobre uma determinada área da linguagem, você definitivamente encontrá neste documento o que estás procurando. Caso queiras ver uma definição mais formal do linguagem, talvez possas oferecer seu tempo — ou inventar uma máquina de clonagem :-).

É perigoso adicionar muitos detalhes de implementação num documento de referência de uma linguagem - a implementação pode mudar e outras implementações da mesma linguagem podem funcionar de forma diferente. Por outro lado, o CPython é a única implementação de Python em uso de forma generalizada (embora as implementações alternativas continuem a ganhar suporte), e suas peculiaridades e particulares são por vezes dignas de serem mencionadas, especialmente quando a implementação impõe limitações adicionais. Portanto, encontrarás poucas "notas sobre a implementação" espalhadas neste documento.

Cada implementação do Python vem com vários módulos internos e por padrão. Estes estão documentados em library-index. Alguns módulos internos são mencionados ao interagirem de forma significativa com a definição da linguagem.

1.1 Implementações Alternativas

Embora exista uma implementação do Python que seja, de longe, a mais popular, existem algumas implementações alternativas que são de de interesse particular e para públicos diferentes.

As implementações conhecidas são:

CPython Esta é a implementação original e a é a versão do Python que mais vem sendo sendo desenvolvido e a mesma está escrita com a linguagem C. Novas funcionalidades ou recursos da linguagem aparecerão por aqui primeiro.

Jython Versão do Python implementado em Java. Esta implementação pode ser usada como linguagem de Script em aplicações Java, ou pode ser usada para criar aplicativos usando as bibliotecas das classes do Java. Também vem sendo bastante utilizado na construção de testes unitários para as bibliotecas do Java. Mais informações podem ser encontradas no the Jython website.

Python for .NET Essa implementação utiliza de fato a implementação CPython, mas é um aplicativo gerenciado .NET e disponibilizado como uma bibliotecas .NET. Foi desenvolvido por Brian Lloyd. Para obter mais informações, consulte Python for .NET home page.

IronPython Um versão alternativa do Python para a plataforma .NET. Ao contrário do Python.NET, esta é uma implementação completa do Python que gera IL e compila o código Python diretamente para assemblies .NET. Foi desenvolvida por Jim Hugunin, o criador original do Jython. Para obter mais informações, consulte the IronPython website.

PyPy Uma implementação do Python escrita completamente em Python. A mesma suporta vários recursos avançados não encontrados em outras implementações, como suporte sem pilhas e um compilador Just in Time. Um dos objetivos do projeto é incentivar a construção de experimentos com a própria linguagem, facilitando a modificação do interpretador (uma vez que o mesmos está escrito em Python). Informações adicionais estão disponíveis na página the PyPy project's home page.

Cada uma dessas implementações varia em alguma forma a linguagem conforme documentado neste manual, ou introduz informações específicas além do que está coberto na documentação padrão do Python. Consulte a documentação específica da implementação para determinar o que é necessário sobre a implementação específica que você está usando.

1.2 Notação

As descrições da Análise Léxica e da Sintaxe usam uma notação de gramática BNF modificada. Isso usa o seguinte estilo de definição:

```
name        ::=   lc_letter (lc_letter | "_")*
lc_letter   ::=   "a"..."z"
```

A primeira linha diz que um `nome` é um `lc_letter` seguido de uma seqüência de zero ou mais `lc_letter``s e underscores. Um ``lc_letter` por sua vez é qualquer um dos caracteres simples `'a'` através de `'z'`. (Esta regra é aderida pelos nomes definidos nas regras léxicas e gramáticas deste documento.)

Cada regra começa com um nome (no caso, o nome definido pela regra) e `::=`. Uma barra vertical (`|`) é usada para separar alternativas; o mesmo é o operador menos vinculativo nesta notação. Uma estrela (`*`) significa zero ou mais repetições do item anterior; da mesma forma, o sinal de adição (+) significa uma ou mais repetições, e uma frase entre colchetes (`[]`) significa zero ou uma ocorrência (em outras palavras, a frase anexada é opcional). Os operadores `*` e + se ligam tão forte quanto possível; parêntesis são usados para o agrupamento. Os literais Strings são delimitados por aspas. O espaço em branco só é significativo para separar os tokens. As regras normalmente estão contidas numa única linha; as regras com muitas alternativas podem ser formatadas alternativamente com cada linha após o primeiro começo com uma barra vertical.

Nas definições léxicas (como o exemplo acima), são utilizadas mais duas convenções: dois caracteres literais separados por três pontos significam a escolha de qualquer caractere único na faixa (inclusiva) fornecida pelos caracteres ASCII. Uma frase entre colchetes angulares (`<...>`) fornece uma descrição informal do símbolo definido; por exemplo, isso poderia ser usado para descrever a notação de 'caractere de controle', caso fosse necessário.

Embora a notação utilizada seja quase a mesma, há uma grande diferença entre o significado das definições lexicais e sintáticas: uma definição lexical opera nos caracteres individuais da fonte de entrada, enquanto uma definição de sintaxe opera no fluxo de tokens gerados pelo analisador lexico. Todos os usos do BNF no próximo capítulo ("Lexical Analysis") são definições léxicas; os usos nos capítulos subsequentes são definições sintáticas.

CAPÍTULO 2

Análise Léxica

Um programa Python é lido por um *analisador*. A entrada para o analisador é um fluxo de *tokens*, gerado pelo *analisador lexical*. Este capítulo descreve como o analisador lexical divide um arquivo em tokens.

Python lê o texto do programa como pontos de código Unicode; a codificação de um arquivo de origem pode ser fornecida por uma declaração de codificação que por padrão é UTF-8, consulte **PEP 3120** para obter detalhes. Se o arquivo de origem não puder ser decodificado, um: exc:*SyntaxError* será gerado.

2.1 Estrutura da Linha

Um programa Python é dividido em uma série de *linhas lógicas*.

2.1.1 Linhas Lógicas

O fim de uma linha lógica é representado pelo token NEWLINE. As declarações não podem cruzar os limites da linha lógica, exceto onde NEWLINE for permitido pela sintaxe (por exemplo, entre as declarações de declarações compostas). Uma linha lógica é construída a partir de uma ou mais *linhas físicas* seguindo a linha explícita ou implícita que *junta as linhas* das regras.

2.1.2 Linha Física

Uma linha física é uma sequência de caracteres terminados por uma sequência de quebra de linha. Nos arquivos de origem e cadeias de caracteres, qualquer uma das sequências de terminação de linha de plataforma padrão pode ser usada - o formato Unix usando ASCII LF (linefeed), o formato Windows usando a sequência ASCII CR LF (return followed by linefeed) ou o antigo formato Macintosh usando o caractere ASCII CR (return). Todos esses formatos podem ser usados igualmente, independentemente da plataforma. O final da entrada também serve como um finalizador implícito para a linha física final.

Ao incorporar o Python, o código-fonte das Strings devem ser passadas para APIs do Python usando as convenções C padrão para caracteres de nova linha (o caractere `\n`, representando ASCII LF, será o terminador de linha).

2.1.3 Comentários

A comment starts with a hash character (#) that is not part of a string literal, and ends at the end of the physical line. A comment signifies the end of the logical line unless the implicit line joining rules are invoked. Comments are ignored by the syntax; they are not tokens.

2.1.4 Declarações de codificação

If a comment in the first or second line of the Python script matches the regular expression `coding[=:]\s*([-\w.]+)`, this comment is processed as an encoding declaration; the first group of this expression names the encoding of the source code file. The encoding declaration must appear on a line of its own. If it is the second line, the first line must also be a comment-only line. The recommended forms of an encoding expression are

```
# -*- coding: <encoding-name> -*-
```

que é reconhecido também por GNU Emacs, e:

```
# vim:fileencoding=<encoding-name>
```

que é reconhecido pelo VIM de Bram Moolenaar.

If no encoding declaration is found, the default encoding is UTF-8. In addition, if the first bytes of the file are the UTF-8 byte-order mark (`b'\xef\xbb\xbf'`), the declared file encoding is UTF-8 (this is supported, among others, by Microsoft's **notepad**).

If an encoding is declared, the encoding name must be recognized by Python. The encoding is used for all lexical analysis, including string literals, comments and identifiers.

2.1.5 Junção de linha explícita

Two or more physical lines may be joined into logical lines using backslash characters (\), as follows: when a physical line ends in a backslash that is not part of a string literal or comment, it is joined with the following forming a single logical line, deleting the backslash and the following end-of-line character. For example:

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:   # Looks like a valid date
        return 1
```

A line ending in a backslash cannot carry a comment. A backslash does not continue a comment. A backslash does not continue a token except for string literals (i.e., tokens other than string literals cannot be split across physical lines using a backslash). A backslash is illegal elsewhere on a line outside a string literal.

2.1.6 Junção de linha implicita

Expressions in parentheses, square brackets or curly braces can be split over more than one physical line without using backslashes. For example:

```
month_names = ['Januari', 'Februari', 'Maart',      # These are the
               'April',   'Mei',      'Juni',       # Dutch names
               'Juli',    'Augustus', 'September',  # for the months
               'Oktober', 'November', 'December']   # of the year
```

Implicitly continued lines can carry comments. The indentation of the continuation lines is not important. Blank continuation lines are allowed. There is no NEWLINE token between implicit continuation lines. Implicitly continued lines can also occur within triple-quoted strings (see below); in that case they cannot carry comments.

2.1.7 Linhas em Branco

A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is ignored (i.e., no NEWLINE token is generated). During interactive input of statements, handling of a blank line may differ depending on the implementation of the read-eval-print loop. In the standard interactive interpreter, an entirely blank logical line (i.e. one containing not even whitespace or a comment) terminates a multi-line statement.

2.1.8 Indentação

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements.

Tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to and including the replacement is a multiple of eight (this is intended to be the same rule as used by Unix). The total number of spaces preceding the first non-blank character then determines the line's indentation. Indentation cannot be split over multiple physical lines using backslashes; the whitespace up to the first backslash determines the indentation.

Indentation is rejected as inconsistent if a source file mixes tabs and spaces in a way that makes the meaning dependent on the worth of a tab in spaces; a `TabError` is raised in that case.

Cross-platform compatibility note: because of the nature of text editors on non-UNIX platforms, it is unwise to use a mixture of spaces and tabs for the indentation in a single source file. It should also be noted that different platforms may explicitly limit the maximum indentation level.

A formfeed character may be present at the start of the line; it will be ignored for the indentation calculations above. Formfeed characters occurring elsewhere in the leading whitespace have an undefined effect (for instance, they may reset the space count to zero).

The indentation levels of consecutive lines are used to generate INDENT and DEDENT tokens, using a stack, as follows.

Before the first line of the file is read, a single zero is pushed on the stack; this will never be popped off again. The numbers pushed on the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, it is pushed on the stack, and one INDENT token is generated. If it is smaller, it *must* be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a DEDENT token is generated. At the end of the file, a DEDENT token is generated for each number remaining on the stack that is larger than zero.

Here is an example of a correctly (though confusingly) indented piece of Python code:

```
def perm(l):
        # Compute the list of all permutations of l
    if len(l) <= 1:
                  return [l]
    r = []
    for i in range(len(l)):
             s = l[:i] + l[i+1:]
             p = perm(s)
             for x in p:
              r.append(l[i:i+1] + x)
    return r
```

The following example shows various indentation errors:

```
 def perm(l):                        # error: first line indented
for i in range(len(l)):              # error: not indented
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])   # error: unexpected indent
        for x in p:
                r.append(l[i:i+1] + x)
            return r                 # error: inconsistent dedent
```

(Actually, the first three errors are detected by the parser; only the last error is found by the lexical analyzer — the indentation of `return r` does not match a level popped off the stack.)

2.1.9 Espaços em branco entre tokens

Except at the beginning of a logical line or in string literals, the whitespace characters space, tab and formfeed can be used interchangeably to separate tokens. Whitespace is needed between two tokens only if their concatenation could otherwise be interpreted as a different token (e.g., ab is one token, but a b is two tokens).

2.2 Outros Tokens

Besides NEWLINE, INDENT and DEDENT, the following categories of tokens exist: *identifiers*, *keywords*, *literals*, *operators*, and *delimiters*. Whitespace characters (other than line terminators, discussed earlier) are not tokens, but serve to delimit tokens. Where ambiguity exists, a token comprises the longest possible string that forms a legal token, when read from left to right.

2.3 Identificadores e Keywords

Identifiers (also referred to as *names*) are described by the following lexical definitions.

The syntax of identifiers in Python is based on the Unicode standard annex UAX-31, with elaboration and changes as defined below; see also **PEP 3131** for further details.

Within the ASCII range (U+0001..U+007F), the valid characters for identifiers are the same as in Python 2.x: the uppercase and lowercase letters `A` through `Z`, the underscore _ and, except for the first character, the digits `0` through `9`.

Python 3.0 introduces additional characters from outside the ASCII range (see **PEP 3131**). For these characters, the classification uses the version of the Unicode Character Database as included in the `unicodedata` module.

Identifiers are unlimited in length. Case is significant.

```
identifier    ::=  xid_start xid_continue*
id_start      ::=  <all characters in general categories Lu, Ll, Lt, Lm, Lo, Nl, the under
id_continue   ::=  <all characters in id_start, plus characters in the categories Mn, Mc,
xid_start     ::=  <all characters in id_start whose NFKC normalization is in "id_start xi
xid_continue  ::=  <all characters in id_continue whose NFKC normalization is in "id_conti
```

The Unicode category codes mentioned above stand for:

- *Lu* - letras maiúsculas
- *Ll* - letras minúsculas
- *Lt* - letras de titlecase

- *Lm* - letras modificadoras
- *Lo* - outras letras
- *Nl* - letras numéricas
- *Mn* - marcas sem espaçamentos
- *Mc* - marcas de combinação de espaçamento
- *Nd* - números decimais
- *Pc* - pontuações de conectores
- *Other_ID_Start* - explicit list of characters in PropList.txt to support backwards compatibility
- *Other_ID_Continue* - likewise

All identifiers are converted into the normal form NFKC while parsing; comparison of identifiers is based on NFKC.

A non-normative HTML file listing all valid identifier characters for Unicode 4.1 can be found at https://www.dcl.hpi.uni-potsdam.de/home/loewis/table-3131.html.

2.3.1 Keywords

The following identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

2.3.2 Classes reservadas de identificadores

Certas classes de identificadores (além de keywords) possuem significados especiais. Essas classes são identificadas pelos padrões de caracteres de sublinhado principais e à direita:

_* Not imported by `from module import *`. The special identifier _ is used in the interactive interpreter to store the result of the last evaluation; it is stored in the `builtins` module. When not in interactive mode, _ has no special meaning and is not defined. See section *The import statement*.

> **Nota:** The name _ is often used in conjunction with internationalization; refer to the documentation for the `gettext` module for more information on this convention.

__*__ System-defined names. These names are defined by the interpreter and its implementation (including the standard library). Current system names are discussed in the *Nomes de métodos especiais* section and elsewhere. More will likely be defined in future versions of Python. *Any* use of __*__ names, in any context, that does not follow explicitly documented use, is subject to breakage without warning.

__* Class-private names. Names in this category, when used within the context of a class definition, are re-written to use a mangled form to help avoid name clashes between "private" attributes of base and derived classes. See section *Identificadores (Nomes)*.

2.4 Literais

Os literais são notações para valores constantes de alguns tipos incorporados.

2.4.1 String e Bytes literais

String literals are described by the following lexical definitions:

```
stringliteral   ::=  [stringprefix](shortstring | longstring)
stringprefix    ::=  "r" | "u" | "R" | "U" | "f" | "F"
                     | "fr" | "Fr" | "fR" | "FR" | "rf" | "rF" | "Rf" | "RF"
shortstring     ::=  "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring      ::=  "'''" longstringitem* "'''" | '"""' longstringitem* '"""'
shortstringitem ::=  shortstringchar | stringescapeseq
longstringitem  ::=  longstringchar | stringescapeseq
shortstringchar ::=  <any source character except "\" or newline or the quote>
longstringchar  ::=  <any source character except "\">
stringescapeseq ::=  "\" <any source character>
```

```
bytesliteral    ::=  bytesprefix(shortbytes | longbytes)
bytesprefix     ::=  "b" | "B" | "br" | "Br" | "bR" | "BR" | "rb" | "rB" | "Rb" | "RB"
shortbytes      ::=  "'" shortbytesitem* "'" | '"' shortbytesitem* '"'
longbytes       ::=  "'''" longbytesitem* "'''" | '"""' longbytesitem* '"""'
shortbytesitem  ::=  shortbyteschar | bytesescapeseq
longbytesitem   ::=  longbyteschar | bytesescapeseq
shortbyteschar  ::=  <any ASCII character except "\" or newline or the quote>
longbyteschar   ::=  <any ASCII character except "\">
bytesescapeseq  ::=  "\" <any ASCII character>
```

One syntactic restriction not indicated by these productions is that whitespace is not allowed between the *stringprefix* or *bytesprefix* and the rest of the literal. The source character set is defined by the encoding declaration; it is UTF-8 if no encoding declaration is given in the source file; see section *Declarações de codificação*.

In plain English: Both types of literals can be enclosed in matching single quotes (`'`) or double quotes (`"`). They can also be enclosed in matching groups of three single or double quotes (these are generally referred to as *triple-quoted strings*). The backslash (`\`) character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character.

Bytes literals are always prefixed with `'b'` or `'B'`; they produce an instance of the `bytes` type instead of the `str` type. They may only contain ASCII characters; bytes with a numeric value of 128 or greater must be expressed with escapes.

Both string and bytes literals may optionally be prefixed with a letter `'r'` or `'R'`; such strings are called *raw strings* and treat backslashes as literal characters. As a result, in string literals, `'\U'` and `'\u'` escapes in raw strings are not treated specially. Given that Python 2.x's raw unicode literals behave differently than Python 3.x's the `'ur'` syntax is not supported.

Novo na versão 3.3: The `'rb'` prefix of raw bytes literals has been added as a synonym of `'br'`.

Novo na versão 3.3: Support for the unicode legacy literal (`u'value'`) was reintroduced to simplify the maintenance of dual Python 2.x and 3.x codebases. See **PEP 414** for more information.

A string literal with `'f'` or `'F'` in its prefix is a *formatted string literal*; see *Literais de string formatados*. The `'f'` may be combined with `'r'`, but not with `'b'` or `'u'`, therefore raw formatted strings are possible, but formatted bytes

literals are not.

In triple-quoted literals, unescaped newlines and quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the literal. (A "quote" is the character used to open the literal, i.e. either `'` or `"`.)

Unless an `'r'` or `'R'` prefix is present, escape sequences in string and bytes literals are interpreted according to rules similar to those used by Standard C. The recognized escape sequences are:

Caracteres de Escapes (Escapes Sequence)	Significado	Notas
`\newline`	A barra invertida e a nova linha foram ignoradas	
`\\`	Backslash (`\`)	
`\'`	Aspas Simples (`'`)	
`\"`	Aspas Dupla (`"`)	
`\a`	ASCII Bell (BEL)	
`\b`	ASCII Backspace (BS)	
`\f`	ASCII Formfeed (FF)	
`\n`	ASCII Linefeed (LF)	
`\r`	ASCII Carriage Return (CR)	
`\t`	ASCII Horizontal Tab (TAB)	
`\v`	ASCII Vertical Tab (VT)	
`\ooo`	Caráter com valor octal *ooo*	(1,3)
`\xhh`	Caráter com valor hexadecimal *hh*	(2,3)

As sequências de escape apenas reconhecidas em literais de Strings são:

Caracteres de Escapes (Escapes Sequence)	Significado	Notas
`\N{name}`	Caractere chamado *name* no banco de dados Unicode	(4)
`\uxxxx`	Caractere com valor hexadecimal de 16 bits *xxxx*	(5)
`\Uxxxxxxxx`	Caractere com valor hexadecimal de 32 bits *xxxxxxxx*	(6)

Notas:

(1) Como no padrão C, são aceitos até três dígitos octal.

(2) Ao contrário do padrão C, são necessários exatamente dois dígitos hexadecimais.

(3) In a bytes literal, hexadecimal and octal escapes denote the byte with the given value. In a string literal, these escapes denote a Unicode character with the given value.

(4) Alterado na versão 3.3: O suporte para apelidos de nome[1] foram adicionado.

(5) São necessários quatro dígitos hexadecimais.

(6) Any Unicode character can be encoded this way. Exactly eight hex digits are required.

Unlike Standard C, all unrecognized escape sequences are left in the string unchanged, i.e., *the backslash is left in the result.* (This behavior is useful when debugging: if an escape sequence is mistyped, the resulting output is more easily recognized as broken.) It is also important to note that the escape sequences only recognized in string literals fall into the category of unrecognized escapes for bytes literals.

> Alterado na versão 3.6: Unrecognized escape sequences produce a DeprecationWarning. In some future version of Python they will be a SyntaxError.

Even in a raw literal, quotes can be escaped with a backslash, but the backslash remains in the result; for example, `r"\""` is a valid string literal consisting of two characters: a backslash and a double quote; `r"\"` is not a valid string literal (even a raw string cannot end in an odd number of backslashes). Specifically, *a raw literal cannot end in a single backslash*

[1] http://www.unicode.org/Public/9.0.0/ucd/NameAliases.txt

(since the backslash would escape the following quote character). Note also that a single backslash followed by a newline is interpreted as those two characters as part of the literal, *not* as a line continuation.

2.4.2 String literal concatenation

Multiple adjacent string or bytes literals (delimited by whitespace), possibly using different quoting conventions, are allowed, and their meaning is the same as their concatenation. Thus, `"hello" 'world'` is equivalent to `"helloworld"`. This feature can be used to reduce the number of backslashes needed, to split long strings conveniently across long lines, or even to add comments to parts of strings, for example:

```
re.compile("[A-Za-z_]"       # letter or underscore
           "[A-Za-z0-9_]*"   # letter, digit or underscore
          )
```

Note that this feature is defined at the syntactical level, but implemented at compile time. The '+' operator must be used to concatenate string expressions at run time. Also note that literal concatenation can use different quoting styles for each component (even mixing raw strings and triple quoted strings), and formatted string literals may be concatenated with plain string literals.

2.4.3 Literais de string formatados

Novo na versão 3.6.

A *formatted string literal* or *f-string* is a string literal that is prefixed with `'f'` or `'F'`. These strings may contain replacement fields, which are expressions delimited by curly braces `{}`. While other string literals always have a constant value, formatted strings are really expressions evaluated at run time.

Escape sequences are decoded like in ordinary string literals (except when a literal is also marked as a raw string). After decoding, the grammar for the contents of the string is:

```
f_string           ::=  (literal_char | "{{" | "}}" | replacement_field)*
replacement_field  ::=  "{" f_expression ["!" conversion] [":" format_spec] "}"
f_expression       ::=  (conditional_expression | "*" or_expr)
                        ("," conditional_expression | "," "*" or_expr)* [","]
                        | yield_expression
conversion         ::=  "s" | "r" | "a"
format_spec        ::=  (literal_char | NULL | replacement_field)*
literal_char       ::=  <any code point except "{", "}" or NULL>
```

The parts of the string outside curly braces are treated literally, except that any doubled curly braces `'{{'` or `'}}'` are replaced with the corresponding single curly brace. A single opening curly bracket `'{'` marks a replacement field, which starts with a Python expression. After the expression, there may be a conversion field, introduced by an exclamation point `'!'`. A format specifier may also be appended, introduced by a colon `':'`. A replacement field ends with a closing curly bracket `'}'`.

Expressions in formatted string literals are treated like regular Python expressions surrounded by parentheses, with a few exceptions. An empty expression is not allowed, and a *lambda* expression must be surrounded by explicit parentheses. Replacement expressions can contain line breaks (e.g. in triple-quoted strings), but they cannot contain comments. Each expression is evaluated in the context where the formatted string literal appears, in order from left to right.

An *await* expression and comprehensions containing an *async for* clause are illegal in the expression in formatted string literals. (The reason is a problem with the implementation — this restriction is lifted in Python 3.7).

If a conversion is specified, the result of evaluating the expression is converted before formatting. Conversion `'!s'` calls

`str()` on the result, `'!r'` calls `repr()`, and `'!a'` calls `ascii()`.

The result is then formatted using the `format()` protocol. The format specifier is passed to the *__format__()* method of the expression or conversion result. An empty string is passed when the format specifier is omitted. The formatted result is then included in the final value of the whole string.

Top-level format specifiers may include nested replacement fields. These nested fields may include their own conversion fields and format specifiers, but may not include more deeply-nested replacement fields. The format specifier mini-language is the same as that used by the string .format() method.

Formatted string literals may be concatenated, but replacement fields cannot be split across literals.

Alguns exemplos de literais formatados:

```
>>> name = "Fred"
>>> f"He said his name is {name!r}."
"He said his name is 'Fred'."
>>> f"He said his name is {repr(name)}."  # repr() is equivalent to !r
"He said his name is 'Fred'."
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}"  # nested fields
'result:      12.35'
>>> today = datetime(year=2017, month=1, day=27)
>>> f"{today:%B %d, %Y}"  # using date format specifier
'January 27, 2017'
>>> number = 1024
>>> f"{number:#0x}"  # using integer format specifier
'0x400'
```

A consequence of sharing the same syntax as regular string literals is that characters in the replacement fields must not conflict with the quoting used in the outer formatted string literal:

```
f"abc {a["x"]} def"    # error: outer string literal ended prematurely
f"abc {a['x']} def"    # workaround: use different quoting
```

As barras invertidas não são permitidas nas expressões de formatação e levantarão uma exceção:

```
f"newline: {ord('\n')}"  # raises SyntaxError
```

To include a value in which a backslash escape is required, create a temporary variable.

```
>>> newline = ord('\n')
>>> f"newline: {newline}"
'newline: 10'
```

Formatted string literals cannot be used as docstrings, even if they do not include expressions.

```
>>> def foo():
...     f"Not a docstring"
...
>>> foo.__doc__ is None
True
```

See also **PEP 498** for the proposal that added formatted string literals, and `str.format()`, which uses a related format string mechanism.

2.4.4 Literais Numéricos

There are three types of numeric literals: integers, floating point numbers, and imaginary numbers. There are no complex literals (complex numbers can be formed by adding a real number and an imaginary number).

Note that numeric literals do not include a sign; a phrase like `-1` is actually an expression composed of the unary operator '`-`' and the literal `1`.

2.4.5 Inteiros Literais

Integer literals are described by the following lexical definitions:

```
integer       ::=   decinteger | bininteger | octinteger | hexinteger
decinteger    ::=   nonzerodigit (["_"] digit)* | "0"+ (["_"] "0")*
bininteger    ::=   "0" ("b" | "B") (["_"] bindigit)+
octinteger    ::=   "0" ("o" | "O") (["_"] octdigit)+
hexinteger    ::=   "0" ("x" | "X") (["_"] hexdigit)+
nonzerodigit  ::=   "1"..."9"
digit         ::=   "0"..."9"
bindigit      ::=   "0" | "1"
octdigit      ::=   "0"..."7"
hexdigit      ::=   digit | "a"..."f" | "A"..."F"
```

There is no limit for the length of integer literals apart from what can be stored in available memory.

Underscores are ignored for determining the numeric value of the literal. They can be used to group digits for enhanced readability. One underscore can occur between digits, and after base specifiers like `0x`.

Note that leading zeros in a non-zero decimal number are not allowed. This is for disambiguation with C-style octal literals, which Python used before version 3.0.

Alguns exemplos de inteiros literais:

```
7     2147483647                        0o177    0b100110111
3     79228162514264337593543950336     0o377    0xdeadbeef
      100_000_000_000                   0b_1110_0101
```

Alterado na versão 3.6: Os underscores agora são permitidos para fins de agrupamento de literais.

2.4.6 Literais de Ponto Flutuante

Floating point literals are described by the following lexical definitions:

```
floatnumber   ::=   pointfloat | exponentfloat
pointfloat    ::=   [digitpart] fraction | digitpart "."
exponentfloat ::=   (digitpart | pointfloat) exponent
digitpart     ::=   digit (["_"] digit)*
fraction      ::=   "." digitpart
exponent      ::=   ("e" | "E") ["+" | "-"] digitpart
```

Note that the integer and exponent parts are always interpreted using radix 10. For example, `077e010` is legal, and denotes the same number as `77e10`. The allowed range of floating point literals is implementation-dependent. As in

integer literals, underscores are supported for digit grouping.

Alguns exemplos de literais de ponto flutuante:

```
3.14    10.    .001    1e100    3.14e-10    0e0    3.14_15_93
```

Alterado na versão 3.6: Os underscores agora são permitidos para fins de agrupamento de literais.

2.4.7 Literais Imaginários

Os literais imaginários são descritos pelas seguintes definições léxicas:

```
imagnumber  ::=  (floatnumber | digitpart) ("j" | "J")
```

Um literal imaginário produz um número complexo com uma parte real de 0.0. Os números complexos são representados como um par de números de ponto flutuante e têm as mesmas restrições em seu alcance. Para criar um número complexo com uma parte real diferente de zero, adicione um número de ponto flutuante a ele, por exemplo, `(3 + 4j)`. Alguns exemplos de literais imaginários

```
3.14j   10.j    10j     .001j   1e100j   3.14e-10j   3.14_15_93j
```

2.5 Operadores

Os seguintes tokens são operadores:

```
+       -       *       **      /       //      %       @
<<      >>      &       |       ^       ~
<       >       <=      >=      ==      !=
```

2.6 Delimitadores

Os seguintes tokens servem como delimitadores na gramática:

```
(       )       [       ]       {       }
,       :       .       ;       @       =       ->
+=      -=      *=      /=      //=     %=      @=
&=      |=      ^=      >>=     <<=     **=
```

O período também pode ocorrer em literais de ponto flutuante e imaginário. Uma sequência de três períodos tem um significado especial como um literal de elipsis. A segunda metade da lista, os operadores de atribuição aumentada, servem lexicamente como delimitadores, mas também realizam uma operação.

The following printing ASCII characters have special meaning as part of other tokens or are otherwise significant to the lexical analyzer:

```
'       "       #       \
```

The following printing ASCII characters are not used in Python. Their occurrence outside string literals and comments is an unconditional error:

```
$           ?           `
```

Notas de Rodapé

CAPÍTULO 3

Modelo de dados

3.1 Objetos, valores e tipos

Objetos são abstrações do Python para dados. Todo dados em um programa Python, é representados por objetos ou por relações entre objetos.(De certo modo, e em conformidade com o modelo de Von Neumann em "stored program computer," código também é representado por objetos.)

Todo objeto tem uma identidade, um tipo e um valor. A *identidade de um objeto, nunca muda depois de criado; você pode pensar nisso como endereço de objetos em memória. O operador ': keyword: *is*' compara as identidades de dois objetos; a função `id()` retorna um inteiro representando sua identidade.

CPython implementation detail: Para CPython, `id(x)` é o endereço de memória em que `x` é armazenado.

O tipo de um objeto determina as operações que o objeto suporta (por exemplo, "ele tem um comprimento?") e também define os valores possíveis para objetos desse tipo. A função `type()` retorna o tipo de um objeto (que é o próprio objeto). Como sua identidade, o objeto: dfn: *type* também é imutável. [#] _

O *valor* de alguns objetos pode mudar. Objetos cujos valores podem mudar são descritos como *mutáveis*, objetos cujo valor não pode ser mudado uma vez que foram criados são chamados *imutáveis*. (O valor de uma coleção que contém uma referência a um objeto mutável pode mudar quando o valor deste último for mudado; no entanto a coleção é ainda assim considerada imutável, pois a coleção de objetos que contém não pode ser mudada. Então a imutabilidade não é estritamente o mesmo do que não haver mudanças de valor, é mais sutil.) A mutabilidade de um objeto é determinada pelo seu tipo; por exemplo, números, strings e tuplas são imutáveis, enquanto dicionários e listas são mutáveis.

Os objetos nunca são destruídos explicitamente; no entanto, quando eles se tornam inacessíveis, eles podem ser coletados como lixo. Uma implementação tem permissão para adiar a coleta de lixo ou omiti-la completamente – é uma questão de qualidade de implementação como a coleta de lixo é implementada, desde que nenhum objeto seja coletado que ainda esteja acessível.

CPython implementation detail: CPython atualmente usa um esquema de contagem de referência com detecção atrasada (opcional) de lixo ligado ciclicamente, que coleta a maioria dos objetos assim que eles se tornam inacessíveis, mas não é garantido que coletará lixo contendo referências circulares. Veja a documentação do módulo `gc` para informações sobre como controlar a coleta de lixo cíclico. Outras implementações agem de forma diferente e o CPython pode mudar. Não dependa da finalização imediata dos objetos quando eles se tornarem inacessíveis (portanto, você deve sempre fechar os arquivos explicitamente).

Observe que o uso dos recursos de rastreamento ou depuração da implementação pode manter os objetos ativos que normalmente seriam coletáveis. Observe também que capturar uma exceção com uma instrução "*try…except*" pode manter os objetos vivos.

Alguns objetos contêm referências a recursos "externos", como arquivos abertos ou janelas. Entende-se que esses recursos são liberados quando o objeto é coletado como lixo, mas como a coleta de lixo não é garantida, tais objetos também fornecem uma maneira explícita de liberar o recurso externo, geralmente um método `close()`. Os programas são fortemente recomendados para fechar explicitamente esses objetos. A instrução "*try…finally*" e a instrução "*with*" fornecem maneiras convenientes de fazer isso.

Alguns objetos contêm referências a outros objetos; eles são chamados de *contêineres*. Exemplos de contêineres são tuplas, listas e dicionários. As referências fazem parte do valor de um contêiner. Na maioria dos casos, quando falamos sobre o valor de um contêiner, nos referimos aos valores, não às identidades dos objetos contidos; entretanto, quando falamos sobre a mutabilidade de um contêiner, apenas as identidades dos objetos contidos imediatamente estão implícitas. Portanto, se um contêiner imutável (como uma tupla) contém uma referência a um objeto mutável, seu valor muda se esse objeto mutável for alterado.

Os tipos afetam quase todos os aspectos do comportamento do objeto. Até mesmo a importância da identidade do objeto é afetada em algum sentido: para tipos imutáveis, as operações que calculam novos valores podem realmente retornar uma referência a qualquer objeto existente com o mesmo tipo e valor, enquanto para objetos mutáveis isso não é permitido. Por exemplo, após `a = 1; b = 1`, `a` e `b` podem ou não se referir ao mesmo objeto com o valor um, dependendo da implementação, mas após `c = []; d = []`, `c` e `d` têm a garantia de referir-se a duas listas vazias diferentes e únicas. (Observe que `c = d = []` atribui o mesmo objeto para `c` e `d`.)

3.2 A hierarquia de tipos padrão

Abaixo está uma lista dos tipos que são embutidos no Python. Módulos de extensão (escritos em C, Java ou outras linguagens, dependendo da implementação) podem definir tipos adicionais. Versões futuras do Python podem adicionar tipos à hierarquia de tipo (por exemplo, números racionais, matrizes de inteiros armazenadas de forma eficiente, etc.), embora tais adições sejam frequentemente fornecidas por meio da biblioteca padrão.

Algumas das descrições de tipo abaixo contêm um parágrafo listando "atributos especiais". Esses são atributos que fornecem acesso à implementação e não se destinam ao uso geral. Sua definição pode mudar no futuro.

None Este tipo possui um único valor. Existe um único objeto com este valor. Este objeto é acessado através do nome embutido `None`. É usado para significar a ausência de um valor em muitas situações, por exemplo, ele é retornado de funções que não retornam nada explicitamente. Seu valor de verdade é falso.

NotImplemented This type has a single value. There is a single object with this value. This object is accessed through the built-in name `NotImplemented`. Numeric methods and rich comparison methods should return this value if they do not implement the operation for the operands provided. (The interpreter will then try the reflected operation, or some other fallback, depending on the operator.) Its truth value is true.

Veja a documentação implementing-the-arithmetic-operations para mais detalhes.;

Ellipsis Este tipo possui um único valor. Existe um único objeto com este valor. Este objeto é acessado através do literal `...` ou do nome embutido `Ellipsis` (reticências). Seu valor de verdade é true.

numbers.Number Eles são criados por literais numéricos e retornados como resultados por operadores aritméticos e funções aritméticas integradas. Os objetos numéricos são imutáveis; uma vez criado, seu valor nunca muda. Os números do Python são, obviamente, fortemente relacionados aos números matemáticos, mas sujeitos às limitações da representação numérica em computadores.

Python distingue entre inteiros, números de ponto flutuante e números complexos:

numbers.Integral Estes representam elementos do conjunto matemático de inteiros (positivos e negativos).

Existem dois tipos de inteiros:

Integers (`int`)

Eles representam números em um intervalo ilimitado, sujeito apenas à memória (virtual) disponível. Para o propósito de operações de deslocamento e máscara, uma representação binária é assumida e os números negativos são representados em uma variante do complemento de 2 que dá a ilusão de uma string infinita de bits de sinal estendendo-se para a esquerda.

Booleans (`bool`) Estes representam os valores da verdade Falsos e Verdadeiros. Os dois objetos que representam os valores `False` e `True` são os únicos objetos booleanos. O tipo booleano é um subtipo do tipo inteiro, e os valores booleanos se comportam como os valores 0 e 1, respectivamente, em quase todos os contextos, com exceção de que, quando convertidos em uma string, as strings `"False"` ou `"True"` são retornados, respectivamente.

As regras para representação de inteiros têm como objetivo fornecer a interpretação mais significativa das operações de deslocamento e máscara envolvendo inteiros negativos.

`numbers.Real` (`float`) Eles representam números de ponto flutuante de precisão dupla no nível da máquina. Você está à mercê da arquitetura da máquina subjacente (e implementação C ou Java) para o intervalo aceito e tratamento de estouro. Python não oferece suporte a números de ponto flutuante de precisão única; a economia no uso do processador e da memória, que normalmente é o motivo de usá-los, é ofuscada pela sobrecarga do uso de objetos em Python, portanto, não há razão para complicar a linguagem com dois tipos de números de ponto flutuante.

`numbers.Complex` (`complex`) Eles representam números complexos como um par de números de ponto flutuante de precisão dupla no nível da máquina. As mesmas advertências se aplicam aos números de ponto flutuante. As partes reais e imaginárias de um número complexo `z` podem ser recuperadas através dos atributos somente leitura `z.real` e `z.imag`.

Sequências Eles representam conjuntos ordenados finitos indexados por números não negativos. A função embutida `len()` retorna o número de itens de uma sequência. Quando o comprimento de uma sequência é *n*, o conjunto de índices contém os números 0, 1, …, *n*-1. O item *i* da sequência *a* é selecionado por `a[i]`.

Sequências também suportam fatiamento: `a[i:j]` seleciona todos os itens com índice *k* de forma que *i* <= *k* < *j*. Quando usada como expressão, uma fatia é uma sequência do mesmo tipo. Isso implica que o conjunto de índices é renumerado para que comece em 0.

Algumas sequências também suportam "fatiamento estendido" com um terceiro parâmetro de "etapa": `a[i:j:k]` seleciona todos os itens de *a* com índice *x* onde `x = i + n*k`, *n* >= `0` e *i* <= *x* < *j*.

As sequências são distinguidas de acordo com sua mutabilidade:

Sequências Imutáveis Um objeto de um tipo de sequência imutável não pode ser alterado depois de criado. (Se o objeto contiver referências a outros objetos, esses outros objetos podem ser mutáveis e podem ser alterados; no entanto, a coleção de objetos diretamente referenciada por um objeto imutável não pode ser alterada.)

Os tipos a seguir são sequências imutáveis:

Strings Uma string é uma sequência de valores que representam pontos de código Unicode. Todos os pontos de código no intervalo `U+0000 - U+10FFFF` podem ser representados em uma string. Python não tem um tipo `char`; em vez disso, cada ponto de código na string é representado como um objeto string com comprimento `1`. A função embutida `ord()` converte um ponto de código de sua forma de string para um inteiro no intervalo `0 - 10FFFF`; `chr()` converte um inteiro no intervalo `0 - 10FFFF` para o comprimento correspondente do objeto de string `1`. `str.encode()` pode ser usado para converter uma `str` para `bytes` usando a codificação de texto fornecida, e `bytes.decode()` pode ser usado para conseguir o oposto.

Tuplas Os itens de uma tupla são objetos Python arbitrários. Tuplas de dois ou mais itens são formadas por listas de expressões separadas por vírgulas. Uma tupla de um item (um "singleton") pode ser formada afixando uma vírgula a uma expressão (uma expressão por si só não cria uma tupla, já que os parênteses

devem ser usados para agrupamento de expressões). Uma tupla vazia pode ser formada por um par vazio de parênteses.

Bytes Um objeto de bytes é um vetor imutável. Os itens são bytes de 8 bits, representados por inteiros no intervalo 0 <= x < 256. Literais de bytes (como `b'abc'`) e o construtor embutido `bytes()` podem ser usado para criar objetos bytes. Além disso, os objetos bytes podem ser decodificados em strings através do método `decode()`.

Sequências Mutáveis As sequências mutáveis podem ser alteradas após serem criadas. As notações de assinatura e fatiamento podem ser usadas como o destino da atribuição e instruções `del` (*delete*, exclusão).

Atualmente, existem dois tipos de sequência mutável intrínseca:

Listas Os itens de uma lista são objetos Python arbitrários. As listas são formadas colocando uma lista separada por vírgulas de expressões entre colchetes. (Observe que não há casos especiais necessários para formar listas de comprimento 0 ou 1.)

Arrays de Bytes Um objeto bytearray é um vetor mutável. Eles são criados pelo construtor embutido `bytearray()`. Além de serem mutáveis (e, portanto, inalteráveis), os vetores de bytes fornecem a mesma interface e funcionalidade que os objetos imutáveis `bytes`.

O módulo de extensão `array` fornece um exemplo adicional de um tipo de sequência mutável, assim como o módulo `collections`.

Tipo Conjunto Eles representam conjuntos finitos e não ordenados de objetos únicos e imutáveis. Como tal, eles não podem ser indexados por nenhum subscrito. No entanto, eles podem ser iterados, e a função embutida `len()` retorna o número de itens em um conjunto. Os usos comuns para conjuntos são testes rápidos de associação, remoção de duplicatas de uma sequência e computação de operações matemáticas como interseção, união, diferença e diferença simétrica.

Para elementos de conjunto, as mesmas regras de imutabilidade se aplicam às chaves de dicionário. Observe que os tipos numéricos obedecem às regras normais para comparação numérica: se dois números forem iguais (por exemplo, `1` e `1.0`), apenas um deles pode estar contido em um conjunto.

Atualmente, existem dois tipos de conjuntos intrínsecos:

Sets (conjuntos) Eles representam um conjunto mutável. Eles são criados pelo construtor embutido `set()` e podem ser modificados posteriormente por vários métodos, como `add()`.

Frozen sets Eles representam um conjunto imutável. Eles são criados pelo construtor embutido `frozenset()`. Como um frozenset é imutável e *hasheável*, ele pode ser usado novamente como um elemento de outro conjunto, ou como uma chave de dicionário.

Mappings Eles representam conjuntos finitos de objetos indexados por conjuntos de índices arbitrários. A notação subscrito `a[k]` seleciona o item indexado por `k` do mapeamento `a`; isso pode ser usado em expressões e como alvo de atribuições ou instruções `del`. A função embutida `len()` retorna o número de itens em um mapeamento.

Atualmente, há um único tipo de mapeamento intrínseco:

Dicionários Eles representam conjuntos finitos de objetos indexados por valores quase arbitrários. Os únicos tipos de valores não aceitáveis como chaves são os valores que contêm listas ou dicionários ou outros tipos mutáveis que são comparados por valor em vez de por identidade de objeto, o motivo é que a implementação eficiente de dicionários requer que o valor de hash de uma chave permaneça constante. Os tipos numéricos usados para chaves obedecem às regras normais para comparação numérica: se dois números forem iguais (por exemplo, `1` e `1.0`), eles podem ser usados alternadamente para indexar a mesma entrada do dicionário.

Os dicionários são mutáveis; eles podem ser criados pela notação `{...}` (veja a seção *Dictionary displays*).

Os módulos de extensão `dbm.ndbm` e `dbm.gnu` fornecem exemplos adicionais de tipos de mapeamento, assim como o módulo `collections`.

Tipo Callable Estes são os tipos aos quais a operação de chamada de função (veja a seção *Calls*) pode ser aplicada:

Funções Definidas pelo Usuário Um objeto função definido pelo usuário será criado pela definição de função (veja a seção *Definições de função*). A mesma deverá ser invocada com uma lista de argumentos contendo o mesmo número de itens que a lista de parâmetros formais da função.

Atributos Especiais:

Atributo	Significado	
`__doc__`	The function's documentation string, or `None` if unavailable; not inherited by subclasses	Writable
`__name__`	O nome da função	Writable
`__qualname__`	A função *qualified name* Novo na versão 3.3.	Writable
`__module__`	O nome do módulo em que a função foi definida ou `None` se indisponível.	Writable
`__defaults__`	A tuple containing default argument values for those arguments that have defaults, or `None` if no arguments have a default value	Writable
`__code__`	O objeto código que representa o corpo da função compilada.	Writable
`__globals__`	Uma referência ao dicionário que contém as variáveis globais da função — o espaço de nomes global do módulo no qual a função foi definida.	Somente Leitura
`__dict__`	O espaço de nomes que oferece suporte a atributos de função arbitrários.	Writable
`__closure__`	`None` or a tuple of cells that contain bindings for the function's free variables.	Somente Leitura
`__annotations__`	Um dicionário contendo anotações de parâmetros. As chaves do dict são os nomes dos parâmetros e `'return'` para a anotação de retorno, se fornecida.	Writable
`__kwdefaults__`	Um dicionário contendo padrões para parâmetros somente-nomeados.	Writable

A maioria dos atributos rotulados como "Gravável" verifica o tipo do valor atribuído.

Os objetos de função também suportam a obtenção e configuração de atributos arbitrários, que podem ser usados, por exemplo, para anexar metadados a funções. A notação de ponto de atributo regular é usada para obter e definir esses atributos. *Observe que a implementação atual só oferece suporte a atributos de função em funções definidas pelo usuário. Atributos de função embutidas podem ser suportados no futuro.*

Additional information about a function's definition can be retrieved from its code object; see the description of internal types below.

Instância de Métodos Um objeto de método de instância combina uma classe, uma instância de classe e qualquer objeto chamável (normalmente uma função definida pelo usuário).

Atributos especiais somente leitura: `__self__` é o objeto de instância da classe, `__func__` é o objeto de função; `__doc__` é a documentação do método (mesmo que `__func__.__doc__`); `__name__` é o nome do método (mesmo que `__func__.__name__`); `__module__` é o nome do módulo no qual o método foi definido, ou `None` se indisponível.

Os métodos também suportam o acesso (mas não a configuração) dos atributos arbitrários da função no objeto de função subjacente.

Os objetos de método definidos pelo usuário podem ser criados ao se obter um atributo de uma classe (talvez por meio de uma instância dessa classe), se esse atributo for um objeto de função definido pelo usuário ou um objeto de método de classe.

Quando um objeto de método de instância é criado recuperando um objeto de função definido pelo usuário de uma classe por meio de uma de suas instâncias, seu atributo `__self__` é a instância, e o objeto de método é considerado vinculado. O atributo `__func__` do novo método é o objeto da função original.

When a user-defined method object is created by retrieving another method object from a class or instance, the behaviour is the same as for a function object, except that the `__func__` attribute of the new instance is not the original method object but its `__func__` attribute.

Quando um objeto de método de instância é criado recuperando um objeto de método de classe de uma classe ou instância, seu atributo `__self__` é a própria classe, e seu atributo `__func__` é o objeto de função subjacente ao método de classe.

Quando um objeto de método de instância é chamado, a função subjacente (`__func__`) é chamada, inserindo a instância de classe (`__self__`) na frente da lista de argumentos. Por exemplo, quando `C` é uma classe que contém uma definição para uma função `f()`, e `x` é uma instância de `C`, chamando `x.f(1)` é equivalente a chamar `C.f(x, 1)`.

Quando um objeto de método de instância é derivado de um objeto de método de classe, a "instância de classe" armazenada em `__self__` será na verdade a própria classe, de modo que chamar `x.f(1)` ou `C.f(1)` é equivalente a chamar `f(C,1)` sendo `f` a função subjacente.

Observe que a transformação de objeto de função em objeto de método de instância ocorre sempre que o atributo é recuperado da instância. Em alguns casos, uma otimização frutífera é atribuir o atributo a uma variável local e chamar essa variável local. Observe também que essa transformação ocorre apenas para funções definidas pelo usuário; outros objetos chamáveis (e todos os objetos não chamáveis) são recuperados sem transformação. Também é importante observar que as funções definidas pelo usuário que são atributos de uma instância de classe não são convertidas em métodos vinculados; isso *apenas* acontece quando a função é um atributo da classe.

Funções Geradoras A function or method which uses the `yield` statement (see section *A declaração yield*) is called a *generator function*. Such a function, when called, always returns an iterator object which can be used to execute the body of the function: calling the iterator's `iterator.__next__()` method will cause the function to execute until it provides a value using the `yield` statement. When the function executes a `return` statement or falls off the end, a `StopIteration` exception is raised and the iterator will have reached the end of the set of values to be returned.

Coroutine functions Uma função ou um método que é definida(o) usando `async def` é chamado de *função de corrotina*. Tal função, quando chamada, retorna um objeto de *corrotina*. Ele pode conter expressões `await`, bem como instruções `async with` e `async for`. Veja também a seção *Objetos Coroutine*.

Funções geradoras assíncronas Uma função ou um método que é definida(o) usando `async def` e que usa a instrução `yield` é chamada de *função geradora assíncrona*. Tal função, quando chamada, retorna um objeto iterador assíncrono que pode ser usado em uma instrução `async for` para executar o corpo da função.

Chamar o método `aiterator.__anext__()` do iterador assíncrono retornará um *aguardável* que, quando aguardado, será executado até fornecer um valor usando a expressão `yield`. Quando a função executa uma instrução vazia `return` ou cai no final, uma exceção `StopAsyncIteration` é levantada e o iterador assíncrono terá alcançado o final do conjunto de valores a serem produzidos.

Funções Built-in Um objeto função embutida é um wrapper em torno de uma função C. Exemplos de funções embutidas são `len()` e `math.sin()` (`math` é um módulo embutido padrão). O número e o tipo dos argumentos são determinados pela função C. Atributos especiais de somente leitura: `__doc__` é a string de documentação da função, ou `None` se indisponível; `__name__` é o nome da função; `__self__` é definido como `None` (mas veja o próximo item); `__module__` é o nome do módulo no qual a função foi definida ou `None` se indisponível.

Métodos Built-in Este é realmente um disfarce diferente de uma função embutida, desta vez contendo um objeto passado para a função C como um argumento extra implícito. Um exemplo de método embutido é `alist.append()`, presumindo que *alist* é um objeto de lista. Nesse caso, o atributo especial de somente leitura

__self__ é definido como o objeto denotado por *alist*.

Classes Classes são chamáveis. Esses objetos normalmente agem como fábricas para novas instâncias de si mesmos, mas variações são possíveis para tipos de classe que substituem *__new__()*. Os argumentos da chamada são passados para *__new__()* e, no caso típico, para *__init__()* para inicializar a nova instância.

Instância de Classe Instâncias de classes arbitrárias podem ser tornados chamáveis definindo um método *__call__()* em sua classe.

Módulos Modules are a basic organizational unit of Python code, and are created by the *import system* as invoked either by the *import* statement (see *import*), or by calling functions such as importlib.import_module() and built-in __import__(). A module object has a namespace implemented by a dictionary object (this is the dictionary referenced by the __globals__ attribute of functions defined in the module). Attribute references are translated to lookups in this dictionary, e.g., m.x is equivalent to m.__dict__["x"]. A module object does not contain the code object used to initialize the module (since it isn't needed once the initialization is done).

A atribuição de atributo atualiza o dicionário de espaço de nomes do módulo, por exemplo, m.x = 1 é equivalente a m.__dict__["x"] = 1.

Atributos predefinidos (graváveis): *__name__* é o nome do módulo; __doc__ é a string de documentação do módulo, ou None se indisponível; __annotations__ (opcional) é um dicionário contendo *anotações de variáveis* coletadas durante a execução do corpo do módulo; *__file__* é o caminho do arquivo do qual o módulo foi carregado, se ele foi carregado de um arquivo. O atributo *__file__* pode estar faltando para certos tipos de módulos, como módulos C que estão estaticamente vinculados ao interpretador; para módulos de extensão carregados dinamicamente de uma biblioteca compartilhada, é o nome do caminho do arquivo da biblioteca compartilhada.

Atributo especial somente leitura: __dict__ é o espaço de nomes do módulo como um objeto de dicionário.

CPython implementation detail: Por causa da maneira como o CPython limpa os dicionários do módulo, o dicionário do módulo será limpo quando o módulo sair do escopo, mesmo se o dicionário ainda tiver referências ativas. Para evitar isso, copie o dicionário ou mantenha o módulo por perto enquanto usa seu dicionário diretamente.

Classes Personalizadas Tipos de classe personalizados são tipicamente criados por definições de classe (veja a seção *Definições de classe*). Uma classe possui um espaço de nomes implementado por um objeto de dicionário. As referências de atributos de classe são traduzidas para pesquisas neste dicionário, por exemplo, Cx é traduzido para C.__dict__["x"] (embora haja uma série de ganchos que permitem outros meios de localizar atributos) . Quando o nome do atributo não é encontrado lá, a pesquisa do atributo continua nas classes base. Essa pesquisa das classes base usa a ordem de resolução de métodos C3, que se comporta corretamente mesmo na presença de estruturas de herança "diamante", onde há vários caminhos de herança que levam de volta a um ancestral comum. Detalhes adicionais sobre o C3 MRO usado pelo Python podem ser encontrados na documentação que acompanha a versão 2.3 em https://www.python.org/download/releases/2.3/mro/.

Quando uma referência de atributo de classe (para uma classe C, digamos) produziria um objeto de método de classe, ele é transformado em um objeto de método de instância cujo atributo __self__ é C. Quando produziria um objeto de método estático, ele é transformado no objeto encapsulado pelo objeto de método estático. Veja a seção *Implementando descritores* para outra maneira em que os atributos recuperados de uma classe podem diferir daqueles realmente contidos em seu __dict__.

As atribuições de atributos de classe atualizam o dicionário da classe, nunca o dicionário de uma classe base.

Um objeto de classe pode ser chamado (veja acima) para produzir uma instância de classe (veja abaixo).

Atributos especiais: __name__ é o nome da classe; __module__ é o nome do módulo no qual a classe foi definida; __dict__ é o dicionário que contém o espaço de nomes da classe; __bases__ é uma tupla contendo as classes base, na ordem de sua ocorrência na lista de classes base; __doc__ é a string de documentação da classe, ou None se indefinido; __annotations__ (opcional) é um dicionário contendo *anotações de variáveis* coletadas durante a execução do corpo da classe.

Instância de Classe Uma instância de classe é criada chamando um objeto de classe (veja acima). Uma instância de classe tem um espaço de nomes implementado como um dicionário que é o primeiro lugar no qual as referências

de atributos são pesquisadas. Quando um atributo não é encontrado lá, e a classe da instância possui um atributo com esse nome, a pesquisa continua com os atributos da classe. Se for encontrado um atributo de classe que seja um objeto função definido pelo usuário, ele é transformado em um objeto de método de instância cujo atributo `__self__` é a instância. Métodos estáticos e objetos de método de classe também são transformados; veja acima em "Classes". Veja a seção *Implementando descritores* para outra maneira em que os atributos de uma classe recuperados através de suas instâncias podem diferir dos objetos realmente armazenados no `__dict__` da classe. Se nenhum atributo de classe for encontrado, e a classe do objeto tiver um método *`__getattr__()`*, que é chamado para satisfazer a pesquisa.

As atribuições e exclusões de atributos atualizam o dicionário da instância, nunca o dicionário de uma classe. Se a classe tem um método *`__setattr__()`* ou *`__delattr__()`*, ele é chamado ao invés de atualizar o dicionário da instância diretamente.

As instâncias de classe podem fingir ser números, sequências ou mapeamentos se tiverem métodos com certos nomes especiais. Veja a seção *Nomes de métodos especiais*.

Atributos especiais: `__dict__` é o dicionário de atributos; `__class__` é a classe da instância.

Objetos de E/S (também conhecidos como objetos arquivo) O *objeto de arquivo* representa um arquivo aberto. Vários atalhos estão disponíveis para criar objetos arquivos: a função embutida `open()`, e também `os.popen()`, `os.fdopen()` e o método `makefile()` de objetos soquete (e talvez por outras funções ou métodos fornecidos por módulos de extensão).

Os objetos `sys.stdin`, `sys.stdout` e `sys.stderr` são inicializados para objetos arquivo que correspondem aos fluxos de entrada, saída e erro padrão do interpretador; eles são todos abertos em modo texto e, portanto, seguem a interface definida pela classe abstrata `io.TextIOBase`.

Tipos Internos Alguns tipos usados internamente pelo interpretador são expostos ao usuário. Suas definições podem mudar com versões futuras do interpretador, mas são mencionadas aqui para fins de integridade.

Objeto Código Objetos código representam código Python executável *compilados em bytes* ou *bytecode*. A diferença entre um objeto código e um objeto função é que o objeto função contém uma referência explícita aos globais da função (o módulo no qual foi definida), enquanto um objeto código não contém nenhum contexto; também os valores de argumento padrão são armazenados no objeto função, não no objeto código (porque eles representam os valores calculados em tempo de execução). Ao contrário dos objetos função, os objetos código são imutáveis e não contêm referências (direta ou indiretamente) a objetos mutáveis.

Special read-only attributes: `co_name` gives the function name; `co_argcount` is the number of positional arguments (including arguments with default values); `co_nlocals` is the number of local variables used by the function (including arguments); `co_varnames` is a tuple containing the names of the local variables (starting with the argument names); `co_cellvars` is a tuple containing the names of local variables that are referenced by nested functions; `co_freevars` is a tuple containing the names of free variables; `co_code` is a string representing the sequence of bytecode instructions; `co_consts` is a tuple containing the literals used by the bytecode; `co_names` is a tuple containing the names used by the bytecode; `co_filename` is the filename from which the code was compiled; `co_firstlineno` is the first line number of the function; `co_lnotab` is a string encoding the mapping from bytecode offsets to line numbers (for details see the source code of the interpreter); `co_stacksize` is the required stack size (including local variables); `co_flags` is an integer encoding a number of flags for the interpreter.

Os seguintes bits de sinalizador são definidos para `co_flags`: o bit `0x04` é definido se a função usa a sintaxe `*arguments` para aceitar um número arbitrário de argumentos posicionais; o bit `0x08` é definido se a função usa a sintaxe `**keywords` para aceitar argumentos nomeados arbitrários; o bit `0x20` é definido se a função for um gerador.

Declarações de recursos futuros (`from __future__ import division`) também usam bits em `co_flags` para indicar se um objeto código foi compilado com um recurso específico habilitado: o bit `0x2000` é definido se a função foi compilada com divisão futura habilitada; os bits `0x10` e `0x1000` foram usados em versões anteriores do Python.

Outros bits em `co_flags` são reservados para uso interno.

Se um objeto código representa uma função, o primeiro item em `co_consts` é a string de documentação da função, ou `None` se indefinido.

Objetos quadro Frame objects represent execution frames. They may occur in traceback objects (see below).

Atributos especiais de somente leitura: `f_back` é para o quadro de pilha anterior (para o chamador), ou `None` se este é o quadro de pilha inferior; `f_code` é o objeto código sendo executado neste quadro; `f_locals` é o dicionário usado para procurar variáveis locais; `f_globals` é usado para variáveis globais; `f_builtins` é usado para nomes embutidos (intrínsecos); `f_lasti` dá a instrução precisa (este é um índice para a string bytecode do objeto código).

Special writable attributes: `f_trace`, if not `None`, is a function called at the start of each source code line (this is used by the debugger); `f_lineno` is the current line number of the frame — writing to this from within a trace function jumps to the given line (only for the bottom-most frame). A debugger can implement a Jump command (aka Set Next Statement) by writing to f_lineno.

Objetos quadro têm suporte a um método:

`frame.`**`clear`**`()`
Este método limpa todas as referências a variáveis locais mantidas pelo quadro. Além disso, se o quadro pertencer a um gerador, o gerador é finalizado. Isso ajuda a quebrar os ciclos de referência que envolvem objetos quadro (por exemplo, ao capturar uma exceção e armazenar seu traceback para uso posterior).

`RuntimeError` é levantada se o quadro estiver em execução.

Novo na versão 3.4.

Objetos traceback Traceback objects represent a stack trace of an exception. A traceback object is created when an exception occurs. When the search for an exception handler unwinds the execution stack, at each unwound level a traceback object is inserted in front of the current traceback. When an exception handler is entered, the stack trace is made available to the program. (See section *The try statement*.) It is accessible as the third item of the tuple returned by `sys.exc_info()`. When the program contains no suitable handler, the stack trace is written (nicely formatted) to the standard error stream; if the interpreter is interactive, it is also made available to the user as `sys.last_traceback`.

Special read-only attributes: `tb_next` is the next level in the stack trace (towards the frame where the exception occurred), or `None` if there is no next level; `tb_frame` points to the execution frame of the current level; `tb_lineno` gives the line number where the exception occurred; `tb_lasti` indicates the precise instruction. The line number and last instruction in the traceback may differ from the line number of its frame object if the exception occurred in a *`try`* statement with no matching except clause or with a finally clause.

Objetos slice Objetos slice são usados para representar fatias para métodos *`__getitem__()`*. Eles também são criados pela função embutida `slice()`.

Atributos especiais de somente leitura: `start` é o limite inferior; `stop` é o limite superior; `step` é o valor intermediário; cada um é `None` se omitido. Esses atributos podem ter qualquer tipo.

Objetos slice têm suporte a um método:

`slice.`**`indices`**`(`*`self`*`, `*`length`*`)`
Este método usa um único argumento inteiro *length* e calcula informações sobre a fatia que o objeto slice descreveria se aplicado a uma sequência de itens de *length*. Ele retorna uma tupla de três inteiros; respectivamente, estes são os índices *start* e *stop* e o *step* ou comprimento de avanços da fatia. Índices ausentes ou fora dos limites são tratados de maneira consistente com fatias regulares.

Objetos método estáticos Objetos método estáticos fornecem uma maneira de derrotar a transformação de objetos função em objetos método descritos acima. Um objeto método estático é um wrapper em torno de

qualquer outro objeto, geralmente um objeto método definido pelo usuário. Quando um objeto método estático é recuperado de uma classe ou instância de classe, o objeto realmente retornado é o objeto envolto, que não está sujeito a nenhuma transformação posterior. Os objetos método estáticos não são chamáveis, embora os objetos que envolvem geralmente o sejam. Objetos método estáticos são criados pelo construtor embutido `staticmethod()`.

Objetos métodos de classe Um objeto método de classe, como um objeto método estático, é um invólucro em torno de outro objeto que altera a maneira como esse objeto é recuperado de classes e instâncias de classe. O comportamento dos objetos de método de classe após tal recuperação é descrito acima, em "Métodos definidos pelo usuário". Objetos de método de classe são criados pelo construtor embutido `classmethod()`.

3.3 Nomes de métodos especiais

Uma classe pode implementar certas operações que são chamadas por sintaxe especial (como operações aritméticas ou subscript e fatiamento), definindo métodos com nomes especiais. Esta é a abordagem do Python para *sobrecarga de operador*, permitindo que as classes definam seu próprio comportamento em relação aos operadores da linguagem. Por exemplo, se uma classe define um método chamado *__getitem__()*, e `x` é uma instância desta classe, então `x[i]` é aproximadamente equivalente a `type(x).__getitem__(x, i)`. Exceto onde mencionado, as tentativas de executar uma operação levantam uma exceção quando nenhum método apropriado é definido (tipicamente `AttributeError` ou `TypeError`).

Definir um método especial para `None` indica que a operação correspondente não está disponível. Por exemplo, se uma classe define *__iter__()* para `None`, a classe não é iterável, então chamar `iter()` em suas instâncias irá levantar um `TypeError` (sem retroceder para *__getitem__()*).[2]

Ao implementar uma classe que emula qualquer tipo embutido, é importante que a emulação seja implementada apenas na medida em que faça sentido para o objeto que está sendo modelado. Por exemplo, algumas sequências podem funcionar bem com a recuperação de elementos individuais, mas extrair uma fatia pode não fazer sentido. (Um exemplo disso é a interface `NodeList` no Document Object Model do W3C.)

3.3.1 Customização básica

`object.`**`__new__`**`(`*cls*`[, ...])`

Chamado para criar uma nova instância da classe *cls*. *__new__()* é um método estático (é um caso especial, então você não precisa declará-lo como tal) que recebe a classe da qual uma instância foi solicitada como seu primeiro argumento. Os argumentos restantes são aqueles passados para a expressão do construtor do objeto (a chamada para a classe). O valor de retorno de *__new__()* deve ser a nova instância do objeto (geralmente uma instância de *cls*).

Implementações típicas criam uma nova instância da classe invocando o método *__new__()* da superclasse usando `super().__new__(cls[, ...])` com os argumentos apropriados e, em seguida, modificando a instância recém-criada conforme necessário antes de retorná-la.

If *__new__()* returns an instance of *cls*, then the new instance's *__init__()* method will be invoked like `__init__(self[, ...])`, where *self* is the new instance and the remaining arguments are the same as were passed to *__new__()*.

Se *__new__()* não retornar uma instância de *cls*, então o método *__init__()* da nova instância não será invocado.

[2] Os métodos *__hash__()*, *__iter__()*, *__reversed__()* e *__contains__()* têm um tratamento especial para isso; outros ainda irão gerar um `TypeError`, mas podem fazer isso contando com o comportamento de que `None` não pode ser chamado.

__new__() destina-se principalmente a permitir que subclasses de tipos imutáveis (como int, str ou tupla) personalizem a criação de instâncias. Também é comumente substituído em metaclasses personalizadas para personalizar a criação de classes.

`object.`**`__init__`**`(`*self*`[, ...])`
Chamado após a instância ter sido criada (por *__new__()*), mas antes de ser retornada ao chamador. Os argumentos são aqueles passados para a expressão do construtor da classe. Se uma classe base tem um método *__init__()*, o método *__init__()* da classe derivada, se houver, deve chamá-lo explicitamente para garantir a inicialização apropriada da parte da classe base da instância; por exemplo: `super().__init__([args...])`.

Porque *__new__()* e :meth: __init__‘ trabalham juntos na construção de objetos (*__new__()* para criá-lo e *__init__()* para personalizá-lo), nenhum valor diferente de `None` pode ser retornado por *__init__()*; fazer isso fará com que uma `TypeError` seja levantada em tempo de execução.

`object.`**`__del__`**`(`*self*`)`
Chamado quando a instância está prestes a ser destruída. Também é chamada de finalizador ou (incorretamente) de destruidor. Se uma classe base tem um método *__del__()*, o método *__del__()* da classe derivada, se houver, deve chamá-lo explicitamente para garantir a exclusão adequada da parte da classe base da instância.

É possível (embora não recomendado!) para o método *__del__()* adiar a destruição da instância criando uma nova referência a ela. Isso é chamado de *ressurreição* de objeto. Depende da implementação se *__del__()* é chamado uma segunda vez quando um objeto ressuscitado está prestes a ser destruído; a implementação atual do *CPython* apenas o chama uma vez.

Não é garantido que os métodos *__del__()* sejam chamados para objetos que ainda existam quando o interpretador sai.

Nota: `del x` não chama diretamente `x.__del__()` – o primeiro diminui a contagem de referências para `x` em um, e o segundo só é chamado quando a contagem de referências de `x` atinge zero.

CPython implementation detail: It is possible for a reference cycle to prevent the reference count of an object from going to zero. In this case, the cycle will be later detected and deleted by the *cyclic garbage collector*. A common cause of reference cycles is when an exception has been caught in a local variable. The frame's locals then reference the exception, which references its own traceback, which references the locals of all frames caught in the traceback.

Ver também:

Documentação do módulo `gc`.

Aviso: Devido às circunstâncias precárias sob as quais os métodos *__del__()* são invocados, as exceções que ocorrem durante sua execução são ignoradas e um aviso é impresso em `sys.stderr` em seu lugar. Em particular:

- *__del__()* pode ser chamado quando um código arbitrário está sendo executado, incluindo de qualquer thread arbitrário. Se *__del__()* precisar bloquear ou invocar qualquer outro recurso de bloqueio, pode ocorrer um deadlock, pois o recurso já pode ter sido levado pelo código que é interrompido para executar *__del__()*.
- *__del__()* pode ser executado durante o desligamento do interpretador. Como consequência, as variáveis globais que ele precisa acessar (incluindo outros módulos) podem já ter sido excluídas ou definidas como `None`. O Python garante que os globais cujo nome comece com um único sublinhado sejam excluídos de seu módulo antes que outros globais sejam excluídos; se nenhuma outra referência a tais globais existir, isso pode ajudar a garantir que os módulos importados ainda estejam disponíveis no momento em que o método *__del__()* for chamado.

object.**__repr__**(*self*)
Chamado pela função embutida `repr()` para calcular a representação da string "oficial" de um objeto. Se possível, isso deve parecer uma expressão Python válida que pode ser usada para recriar um objeto com o mesmo valor (dado um ambiente apropriado). Se isso não for possível, uma string no formato `<...alguma descrição útil...>` deve ser retornada. O valor de retorno deve ser um objeto string. Se uma classe define *__repr__()*, mas não *__str__()*, então *__repr__()* também é usado quando uma representação de string "informal" de instâncias daquela classe é necessária.

Isso é normalmente usado para depuração, portanto, é importante que a representação seja rica em informações e inequívoca.

object.**__str__**(*self*)
Chamado por `str(object)` e as funções embutidas `format()` e `print()` para calcular a representação da string "informal" ou agradável para exibição de um objeto. O valor de retorno deve ser um objeto string.

Este método difere de *object.__repr__()* por não haver expectativa de que *__str__()* retorne uma expressão Python válida: uma representação mais conveniente ou concisa pode ser usada.

A implementação padrão definida pelo tipo embutido `object` chama *object.__repr__()*.

object.**__bytes__**(*self*)
Chamado por bytes para calcular uma representação de string de bytes de um objeto. Isso deve retornar um objeto `bytes`.

object.**__format__**(*self*, *format_spec*)
Called by the `format()` built-in function, and by extension, evaluation of *formatted string literals* and the `str.format()` method, to produce a "formatted" string representation of an object. The `format_spec` argument is a string that contains a description of the formatting options desired. The interpretation of the `format_spec` argument is up to the type implementing *__format__()*, however most classes will either delegate formatting to one of the built-in types, or use a similar formatting option syntax.

Consulte formatspec para uma descrição da sintaxe de formatação padrão.

O valor de retorno deve ser um objeto string.

Alterado na versão 3.4: O método __format__ do próprio `object` levanta uma `TypeError` se passada qualquer string não vazia.

object.**__lt__**(*self*, *other*)
object.**__le__**(*self*, *other*)
object.**__eq__**(*self*, *other*)
object.**__ne__**(*self*, *other*)
object.**__gt__**(*self*, *other*)
object.**__ge__**(*self*, *other*)
Esses são os chamados métodos de "comparação rica". A correspondência entre os símbolos do operador e os nomes dos métodos é a seguinte: `x<y` chama `x.__lt__(y)`, `x<=y` chama `x.__le__(y)`, `x==y` chama `x.__eq__(y)`, `x!=y` chama `x.__ne__(y)`, `x>y` chama `x.__gt__(y)` e `x>=y` chama `x.__ge__(y)`.

Um método de comparação rico pode retornar o singleton `NotImplemented` se não implementar a operação para um determinado par de argumentos. Por convenção, `False` e `True` são retornados para uma comparação bem-sucedida. No entanto, esses métodos podem retornar qualquer valor, portanto, se o operador de comparação for usado em um contexto booleano (por exemplo, na condição de uma instrução `if`), Python irá chamar `bool()` no valor para determinar se o resultado for verdadeiro ou falso.

Por padrão, *__ne__()* delega para *__eq__()* e inverte o resultado a menos que seja `NotImplemented`. Não há outras relações implícitas entre os operadores de comparação, por exemplo, a verdade de `(x<y or x==y)` não implica `x<=y`. Para gerar operações de ordenação automaticamente a partir de uma única operação raiz, consulte `functools.total_ordering()`.

Veja o parágrafo sobre *__hash__()* para algumas notas importantes sobre a criação de objetos hasheáveis que suportam operações de comparação personalizadas e são utilizáveis como chaves de dicionário.

Não há versões de argumentos trocados desses métodos (a serem usados quando o argumento esquerdo não tem suporta à operação, mas o argumento direito sim); em vez disso, *__lt__()* e *__gt__()* são o reflexo um do outro, *__le__()* e *__ge__()* são o reflexo um do outro, e *__eq__()* e *__ne__()* são seu próprio reflexo. Se os operandos são de tipos diferentes e o tipo do operando direito é uma subclasse direta ou indireta do tipo do operando esquerdo, o método refletido do operando direito tem prioridade, caso contrário, o método do operando esquerdo tem prioridade. A subclasse virtual não é considerada.

`object.`**`__hash__`**(*self*)

Chamado pela função embutida `hash()` e para operações em membros de coleções em hash incluindo `set`, `frozenset` e `dict`. *__hash__()* deve retornar um inteiro. A única propriedade necessária é que os objetos que são comparados iguais tenham o mesmo valor de hash; é aconselhável misturar os valores hash dos componentes do objeto que também desempenham um papel na comparação dos objetos, empacotando-os em uma tupla e fazendo o hash da tupla. Exemplo:

```
def __hash__(self):
    return hash((self.name, self.nick, self.color))
```

Nota: `hash()` trunca o valor retornado do método *__hash__()* personalizado de um objeto para o tamanho de um `Py_ssize_t`. Isso é normalmente 8 bytes em compilações de 64 bits e 4 bytes em compilações de 32 bits. Se o *__hash__()* de um objeto deve interoperar em compilações de tamanhos de bits diferentes, certifique-se de verificar a largura em todas as compilações suportadas. Uma maneira fácil de fazer isso é com `python -c "import sys; print(sys.hash_info.width)"`.

Se uma classe não define um método *__eq__()*, ela também não deve definir uma operação *__hash__()*; se define *__eq__()* mas não *__hash__()*, suas instâncias não serão utilizáveis como itens em coleções hasheáveis. Se uma classe define objetos mutáveis e implementa um método *__eq__()*, ela não deve implementar *__hash__()*, uma vez que a implementação de coleções hasheáveis requer que o valor hash de uma chave seja imutável (se o valor hash do objeto mudar, estará no balde de hash errado).

As classes definidas pelo usuário têm os métodos *__eq__()* e *__hash__()* por padrão; com eles, todos os objetos se comparam desiguais (exceto com eles mesmos) e `x.__hash__()` retorna um valor apropriado tal que `x == y` implica que `x is y` e `hash(x) == hash(y)`.

A class that overrides *__eq__()* and does not define *__hash__()* will have its *__hash__()* implicitly set to `None`. When the *__hash__()* method of a class is `None`, instances of the class will raise an appropriate `TypeError` when a program attempts to retrieve their hash value, and will also be correctly identified as unhashable when checking `isinstance(obj, collections.Hashable)`.

Se uma classe que sobrescreve *__eq__()* precisa manter a implementação de *__hash__()* de uma classe pai, o interpretador deve ser informado disso explicitamente pela configuração `__hash__ = <ClassePai>.__hash__`.

If a class that does not override *__eq__()* wishes to suppress hash support, it should include `__hash__ = None` in the class definition. A class which defines its own *__hash__()* that explicitly raises a `TypeError` would be incorrectly identified as hashable by an `isinstance(obj, collections.Hashable)` call.

Nota: By default, the *__hash__()* values of str, bytes and datetime objects are "salted" with an unpredictable random value. Although they remain constant within an individual Python process, they are not predictable between repeated invocations of Python.

Isso se destina a fornecer proteção contra uma negação de serviço causada por entradas cuidadosamente escolhidas que exploram o pior caso de desempenho de uma inserção de dicionário, complexidade O(n^2). Consulte http://www.ocert.org/advisories/ocert-2011-003.html para obter detalhes.

Changing hash values affects the iteration order of dicts, sets and other mappings. Python has never made guarantees about this ordering (and it typically varies between 32-bit and 64-bit builds).

Consulte também `PYTHONHASHSEED`.

Alterado na versão 3.3: Aleatorização de hash está habilitada por padrão.

`object.__bool__`(*self*)
Chamado para implementar o teste de valor de verdade e a operação embutida `bool()`; deve retornar `False` ou `True`. Quando este método não é definido, *__len__()* é chamado, se estiver definido, e o objeto é considerado verdadeiro se seu resultado for diferente de zero. Se uma classe não define *__len__()* nem *__bool__()*, todas as suas instâncias são consideradas verdadeiras.

3.3.2 Personalizando o acesso aos atributos

Os seguintes métodos podem ser definidos para personalizar o significado do acesso aos atributos (uso, atribuição ou exclusão de `x.name`) para instâncias de classe.

`object.__getattr__`(*self*, *name*)
Chamado quando o acesso padrão ao atributos falha com um `AttributeError` (ou *__getattribute__()* levanta uma `AttributeError` porque *name* não é um atributo de instância ou um atributo na árvore de classes para `self`; ou *__get__()* de uma propriedade *name* levanta `AttributeError`). Este método deve retornar o valor do atributo (calculado) ou levantar uma exceção `AttributeError`.

Observe que se o atributo for encontrado através do mecanismo normal, *__getattr__()* não é chamado. (Esta é uma assimetria intencional entre *__getattr__()* e *__setattr__()*.) Isso é feito tanto por razões de eficiência quanto porque *__getattr__()* não teria como acessar outros atributos da instância. Observe que pelo menos para variáveis de instâncias, você pode fingir controle total não inserindo nenhum valor no dicionário de atributos de instância (mas, em vez disso, inserindo-os em outro objeto). Veja o método *__getattribute__()* abaixo para uma maneira de realmente obter controle total sobre o acesso ao atributo.

`object.__getattribute__`(*self*, *name*)
Chamado incondicionalmente para implementar acessos a atributo para instâncias da classe. Se a classe também define *__getattr__()*, o último não será chamado a menos que *__getattribute__()* o chame explicitamente ou levante um `AttributeError`. Este método deve retornar o valor do atributo (calculado) ou levantar uma exceção `AttributeError`. Para evitar recursão infinita neste método, sua implementação deve sempre chamar o método da classe base com o mesmo nome para acessar quaisquer atributos de que necessita, por exemplo, `object.__getattribute__(self, name)`.

Nota: Este método ainda pode ser ignorado ao procurar métodos especiais como resultado de invocação implícita por meio da sintaxe da linguagem ou funções embutidas. Consulte *Pesquisa de método especial*.

`object.__setattr__`(*self*, *name*, *value*)
Chamado quando uma atribuição de atributo é tentada. Isso é chamado em vez do mecanismo normal (ou seja, armazena o valor no dicionário da instância). *name* é o nome do atributo, *value* é o valor a ser atribuído a ele.

Se *__setattr__()* deseja atribuir a um atributo de instância, ele deve chamar o método da classe base com o mesmo nome, por exemplo, `object.__setattr__(self, name, value)`.

`object.__delattr__`(*self*, *name*)
Como *__setattr__()*, mas para exclusão de atributo em vez de atribuição. Isso só deve ser implementado se `del obj.name` for significativo para o objeto.

`object.__dir__`(*self*)
Chamado quando `dir()` é chamado no objeto. Uma sequência deve ser retornada. `dir()` converte a sequência

retornada em uma lista e a ordena.

Personalizando acesso a atributos de módulos

Para uma personalização mais refinada do comportamento do módulo (definição de atributos, propriedades etc.), pode-se definir o atributo `__class__` de um objeto de módulo para uma subclasse de `types.ModuleType`. Por exemplo:

```
import sys
from types import ModuleType

class VerboseModule(ModuleType):
    def __repr__(self):
        return f'Verbose {self.__name__}'

    def __setattr__(self, attr, value):
        print(f'Setting {attr}...')
        setattr(self, attr, value)

sys.modules[__name__].__class__ = VerboseModule
```

Nota: Setting module `__class__` only affects lookups made using the attribute access syntax – directly accessing the module globals (whether by code within the module, or via a reference to the module's globals dictionary) is unaffected.

Alterado na versão 3.5: O atributo de módulo `__class__` pode agora ser escrito.

Implementando descritores

Os métodos a seguir se aplicam apenas quando uma instância da classe que contém o método (uma classe chamada *descritora*) aparece em uma classe proprietária *owner* (o descritor deve estar no dicionário de classe do proprietário ou no dicionário de classe para um dos seus pais). Nos exemplos abaixo, "o atributo" refere-se ao atributo cujo nome é a chave da propriedade no `__dict__` da classe proprietária.

`object.`**`__get__`**`(`*self*, *instance*, *owner*`)`
: Called to get the attribute of the owner class (class attribute access) or of an instance of that class (instance attribute access). *owner* is always the owner class, while *instance* is the instance that the attribute was accessed through, or `None` when the attribute is accessed through the *owner*. This method should return the (computed) attribute value or raise an `AttributeError` exception.

`object.`**`__set__`**`(`*self*, *instance*, *value*`)`
: Chamado para definir o atributo em uma instância *instance* da classe proprietária para um novo valor, *value*.

`object.`**`__delete__`**`(`*self*, *instance*`)`
: Chamado para excluir o atributo em uma instância *instance* da classe proprietária.

`object.`**`__set_name__`**`(`*self*, *owner*, *name*`)`
: Chamado no momento em que a classe proprietária *owner* é criada. O descritor foi atribuído a *name*.

  Novo na versão 3.6.

O atributo `__objclass__` é interpretado pelo módulo `inspect` como especificando a classe onde este objeto foi definido (configurar isso apropriadamente pode ajudar na introspecção em tempo de execução dos atributos dinâmicos da classe). Para chamáveis, pode indicar que uma instância do tipo fornecido (ou uma subclasse) é esperada ou necessária como o primeiro argumento posicional (por exemplo, CPython define este atributo para métodos não acoplados que são implementados em C).

Invoking Descriptors

Em geral, um descritor é um atributo de objeto com "comportamento de ligação", cujo acesso ao atributo foi substituído por métodos no protocolo do descritor: *__get__()*, *__set__()* e *__delete__()*. Se qualquer um desses métodos for definido para um objeto, é considerado um descritor.

O comportamento padrão para acesso ao atributo é obter, definir ou excluir o atributo do dicionário de um objeto. Por exemplo, `a.x` tem uma cadeia de pesquisa começando com `a.__dict__['x']`, então `type(a).__dict__['x']`, e contando pelas classes base de `type(a)` excluindo metaclasses.

No entanto, se o valor pesquisado for um objeto que define um dos métodos do descritor, o Python pode substituir o comportamento padrão e invocar o método do descritor. Onde isso ocorre na cadeia de precedência depende de quais métodos descritores foram definidos e como eles foram chamados.

O ponto de partida para a invocação do descritor é uma ligação, `a.x`. Como os argumentos são montados depende de `a`:

Chamada direta A chamada mais simples e menos comum é quando o código do usuário invoca diretamente um método descritor: `x.__get__(a)`.

Ligação de instâncias Se estiver ligando a uma instância de objeto, `a.x` é transformado na chamada: `type(a).__dict__['x'].__get__(a, type(a))`.

Ligação de classes Se estiver ligando a uma classe, `A.x` é transformado na chamada: `A.__dict__['x'].__get__(None, A)`.

Ligação de super Se `a` é uma instância de `super`, então a ligação `super(B, obj).m()` procura `obj.__class__.__mro__` para a classe base `A` precedendo imediatamente `B` e, em seguida, invoca o descritor com a chamada: `A.__dict__['m'].__get__(obj, obj.__class__)`.

For instance bindings, the precedence of descriptor invocation depends on the which descriptor methods are defined. A descriptor can define any combination of *__get__()*, *__set__()* and *__delete__()*. If it does not define *__get__()*, then accessing the attribute will return the descriptor object itself unless there is a value in the object's instance dictionary. If the descriptor defines *__set__()* and/or *__delete__()*, it is a data descriptor; if it defines neither, it is a non-data descriptor. Normally, data descriptors define both *__get__()* and *__set__()*, while non-data descriptors have just the *__get__()* method. Data descriptors with *__set__()* and *__get__()* defined always override a redefinition in an instance dictionary. In contrast, non-data descriptors can be overridden by instances.

Os métodos Python (incluindo `staticmethod()` e `classmethod()`) são implementados como descritores sem dados. Assim, as instâncias podem redefinir e substituir métodos. Isso permite que instâncias individuais adquiram comportamentos que diferem de outras instâncias da mesma classe.

A função `property()` é implementada como um descritor de dados. Da mesma forma, as instâncias não podem substituir o comportamento de uma propriedade.

__slots__

__slots__ permite-nos declarar explicitamente membros de dados (como propriedades) e negar a criação de *__dict__* e *__weakref__* (a menos que explicitamente declarado em *__slots__* ou disponível em um pai.)

The space saved over using *__dict__* can be significant.

`object.`**`__slots__`**

Esta variável de classe pode ser atribuída a uma string, iterável ou sequência de strings com nomes de variáveis usados por instâncias. *__slots__* reserva espaço para as variáveis declaradas e evita a criação automática de *__dict__* e *__weakref__* para cada instância.

Observações ao uso de *__slots__*

- Ao herdar de uma classe sem *__slots__*, os atributos *__dict__* e *__weakref__* das instâncias vão sempre ser acessíveis.
- Sem uma variável *__dict__*, as instâncias não podem ser atribuídas a novas variáveis não listadas na definição *__slots__*. As tentativas de atribuir a um nome de variável não listado levantam `AttributeError`. Se a atribuição dinâmica de novas variáveis for desejada, então adicione `'__dict__'` à sequência de strings na declaração *__slots__*.
- Sem uma variável *__weakref__* para cada instância, as classes que definem *__slots__* não suportam referências fracas para suas instâncias. Se for necessário um suporte de referência fraca, adicione `'__weakref__'` à sequência de strings na declaração *__slots__*.
- *__slots__* são implementados no nível de classe criando descritores (*Implementando descritores*) para cada nome de variável. Como resultado, os atributos de classe não podem ser usados para definir valores padrão para variáveis de instância definidas por *__slots__*; caso contrário, o atributo de classe substituiria a atribuição do descritor.
- A ação de uma declaração *__slots__ se limita à classe em que é definida.* *__slots__* declarados nos pais estão disponíveis nas classes infantis. No entanto, as subclasses filhas receberão um *__dict__ * e *__weakref__* a menos que também definam *__slots__* (que deve conter apenas nomes de quaisquer slots *adicionais*).
- Se uma classe define um slot também definido em uma classe base, a variável de instância definida pelo slot da classe base fica inacessível (exceto por recuperar seu descritor diretamente da classe base). Isso torna o significado do programa indefinido. No futuro, uma verificação pode ser adicionada para evitar isso.
- Não vazio *__slots__* não funciona para classes derivadas de tipos embutidos de "comprimento variável", como `int`, `bytes` e `tuple`.
- Qualquer iterável não string pode ser atribuído a *__slots__*. Mapeamentos também podem ser usados; entretanto, no futuro, um significado especial pode ser atribuído aos valores correspondentes a cada chave.
- Atribuição de *__class__* funciona apenas se ambas as classes têm o mesmo *__slots__*.
- A herança múltipla com várias classes pai com slots pode ser usada, mas apenas um pai tem permissão para ter atributos criados por slots (as outras bases devem ter layouts de slots vazios) – violações levantam `TypeError`.

3.3.3 Personalizando a criação de classe

Sempre que uma classe herda de outra classe, *__init_subclass__* é chamado nessa classe. Dessa forma, é possível escrever classes que alteram o comportamento das subclasses. Isso está intimamente relacionado aos decoradores de classe, mas onde decoradores de classe afetam apenas a classe específica à qual são aplicados, `__init_subclass__` aplica-se apenas a futuras subclasses da classe que define o método.

classmethod `object.__init_subclass__`(*cls*)

Este método é chamado sempre que a classe que contém é uma subclasse. *cls* é então a nova subclasse. Se definido como um método de instância normal, esse método é convertido implicitamente em um método de classe.

Argumentos nomeados dados a uma nova classe são passados para a classe pai `__init_subclass__`. Para compatibilidade com outras classes usando `__init_subclass__`, deve-se retirar os argumentos nomeados necessários e passar os outros para a classe base, como em:

```
class Philosopher:
    def __init_subclass__(cls, default_name, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.default_name = default_name
```

(continua na próxima página)

(continuação da página anterior)

```
class AustralianPhilosopher(Philosopher, default_name="Bruce"):
    pass
```

A implementação padrão `object.__init_subclass__` não faz nada, mas levanta um erro se for chamada com quaisquer argumentos.

Nota: A dica da metaclasse `metaclass` é consumida pelo resto da maquinaria de tipo, e nunca é passada para implementações `__init_subclass__`. A metaclasse real (em vez da dica explícita) pode ser acessada como `type(cls)`.

Novo na versão 3.6.

Metaclasses

Por padrão, as classes são construídas usando `type()`. O corpo da classe é executado em um novo espaço de nomes e o nome da classe é vinculado localmente ao resultado de `type(name, bases, namespace)`.

O processo de criação da classe pode ser personalizado passando o argumento nomeado `metaclass` na linha de definição da classe, ou herdando de uma classe existente que incluiu tal argumento. No exemplo a seguir, `MyClass` e `MySubclass` são instâncias de `Meta`:

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

Quaisquer outros argumentos nomeados especificados na definição de classe são transmitidos para todas as operações de metaclasse descritas abaixo.

Quando uma definição de classe é executada, as seguintes etapas ocorrem:

- the appropriate metaclass is determined
- the class namespace is prepared
- the class body is executed
- the class object is created

Determinando a metaclasse apropriada

A metaclasse apropriada para uma definição de classe é determinada da seguinte forma:

- if no bases and no explicit metaclass are given, then `type()` is used
- if an explicit metaclass is given and it is *not* an instance of `type()`, then it is used directly as the metaclass
- if an instance of `type()` is given as the explicit metaclass, or bases are defined, then the most derived metaclass is used

A metaclasse mais derivada é selecionada a partir da metaclasse explicitamente especificada (se houver) e das metaclasses (ou seja, `type(cls)`) de todas as classes básicas especificadas. A metaclasse mais derivada é aquela que é um subtipo

de *todas* essas metaclasses candidatas. Se nenhuma das metaclasses candidatas atender a esse critério, a definição de classe falhará com `TypeError`.

Preparando o espaço de nomes da classe

Once the appropriate metaclass has been identified, then the class namespace is prepared. If the metaclass has a `__prepare__` attribute, it is called as `namespace = metaclass.__prepare__(name, bases, **kwds)` (where the additional keyword arguments, if any, come from the class definition).

Se a metaclasse não tiver o atributo `__prepare__`, então o espaço de nomes da classe é inicializado como um mapeamento ordenado vazio.

Ver também:

PEP 3115 - Metaclasses no Python 3000 Introduzido o gancho de espaço de nomes `__prepare__`

Executando o corpo da classe

O corpo da classe é executado (aproximadamente) como `exec(body, globals(), namespace)`. A principal diferença de uma chamada normal para `exec()` é que o escopo léxico permite que o corpo da classe (incluindo quaisquer métodos) faça referência a nomes dos escopos atual e externo quando a definição de classe ocorre dentro de uma função.

No entanto, mesmo quando a definição de classe ocorre dentro da função, os métodos definidos dentro da classe ainda não podem ver os nomes definidos no escopo da classe. Variáveis de classe devem ser acessadas através do primeiro parâmetro de instância ou métodos de classe, ou através da referência implícita com escopo léxico `__class__` descrita na próxima seção.

Criando o objeto da classe

Uma vez que o espaço de nomes da classe tenha sido preenchido executando o corpo da classe, o objeto da classe é criado chamando `metaclass(name, bases, namespace, **kwds)` (os argumentos adicionais passados aqui são os mesmos passados para `__prepare__`).

Este objeto classe é aquele que será referenciado pela forma de argumento zero de `super()`. `__class__` é uma referência implícita de fechamento criada pelo compilador se qualquer método em um corpo de classe se referir a `__class__` ou `super`. Isso permite que a forma de argumento zero de `super()` identifique corretamente a classe sendo definida com base no escopo léxico, enquanto a classe ou instância que foi usada para fazer a chamada atual é identificada com base no primeiro argumento passado para o método.

CPython implementation detail: In CPython 3.6 and later, the `__class__` cell is passed to the metaclass as a `__classcell__` entry in the class namespace. If present, this must be propagated up to the `type.__new__` call in order for the class to be initialised correctly. Failing to do so will result in a `DeprecationWarning` in Python 3.6, and a `RuntimeError` in Python 3.8.

Ao usar a metaclasse padrão `type`, ou qualquer metaclasse que finalmente chama `type.__new__`, as seguintes etapas de personalização adicionais são invocadas após a criação do objeto classe:

- primeiro, `type.__new__` coleta todos os descritores no espaço de nomes da classe que definem um método *`__set_name__()`*;
- second, all of these `__set_name__` methods are called with the class being defined and the assigned name of that particular descriptor; and
- finalmente, o gancho *`__init_subclass__()`* é chamado no pai imediato da nova classe em sua ordem de resolução de métodos.

Depois que o objeto classe é criado, ele é passado para os decoradores de classe incluídos na definição de classe (se houver) e o objeto resultante é vinculado ao espaço de nomes local como a classe definida.

Quando uma nova classe é criada por `type.__new__`, o objeto fornecido como o parâmetro do espaço de nomes é copiado para um novo mapeamento ordenado e o objeto original é descartado. A nova cópia é envolta em um proxy de somente leitura, que se torna o atributo `__dict__` do objeto classe.

Ver também:

PEP 3135 - Novo super Descreve a referência implícita de fechamento de `__class__`

Usos para metaclasses

Os usos potenciais para metaclasses são ilimitados. Algumas ideias que foram exploradas incluem enum, criação de log, verificação de interface, delegação automática, criação automática de propriedade, proxies, estruturas e bloqueio/sincronização automático/a de recursos.

3.3.4 Personalizando verificações de instância e subclasse

Os seguintes métodos são usados para substituir o comportamento padrão das funções embutidas `isinstance()` e `issubclass()`.

Em particular, a metaclasse `abc.ABCMeta` implementa esses métodos a fim de permitir a adição de classes base abstratas (ABCs) como "classes base virtuais" para qualquer classe ou tipo (incluindo tipos embutidos), incluindo outras ABCs.

`class.__instancecheck__`(*self*, *instance*)
: Retorna verdadeiro se *instance* deve ser considerada uma instância (direta ou indireta) da classe *class*. Se definido, chamado para implementar `isinstance(instance, class)`.

`class.__subclasscheck__`(*self*, *subclass*)
: Retorna verdadeiro se *subclass* deve ser considerada uma subclasse (direta ou indireta) da classe *class*. Se definido, chamado para implementar `issubclass(subclass, class)`.

Observe que esses métodos são pesquisados no tipo (metaclasse) de uma classe. Eles não podem ser definidos como métodos de classe na classe real. Isso é consistente com a pesquisa de métodos especiais que são chamados em instâncias, apenas neste caso a própria instância é uma classe.

Ver também:

PEP 3119 - Introducing Abstract Base Classes Inclui a especificação para personalizar o comportamento de `isinstance()` e `issubclass()` através de *`__instancecheck__()`* e *`__subclasscheck__()`*, com motivação para esta funcionalidade no contexto da adição de classes base abstratas (veja o módulo `abc`) para a linguagem.

3.3.5 Emulando objetos chamáveis

`object.__call__`(*self*[, *args...*])
: Chamado quando a instância é "chamada" como uma função; se este método for definido, `x(arg1, arg2, ...)` é uma abreviatura para `x.__call__(arg1, arg2, ...)`.

3.3.6 Emulando de tipos contêineres

The following methods can be defined to implement container objects. Containers usually are sequences (such as lists or tuples) or mappings (like dictionaries), but can represent other containers as well. The first set of methods is used either to emulate a sequence or to emulate a mapping; the difference is that for a sequence, the allowable keys should be the integers *k* for which `0 <= k < N` where *N* is the length of the sequence, or slice objects, which define a range of items. It is also recommended that mappings provide the methods `keys()`, `values()`, `items()`, `get()`, `clear()`, `setdefault()`, `pop()`, `popitem()`, `copy()`, and `update()` behaving similar to those for Python's standard dictionary objects. The `collections` module provides a `MutableMapping` abstract base class to help create those methods from a base set of *__getitem__()*, *__setitem__()*, *__delitem__()*, and `keys()`. Mutable sequences should provide methods `append()`, `count()`, `index()`, `extend()`, `insert()`, `pop()`, `remove()`, `reverse()` and `sort()`, like Python standard list objects. Finally, sequence types should implement addition (meaning concatenation) and multiplication (meaning repetition) by defining the methods *__add__()*, *__radd__()*, *__iadd__()*, *__mul__()*, *__rmul__()* and *__imul__()* described below; they should not define other numerical operators. It is recommended that both mappings and sequences implement the *__contains__()* method to allow efficient use of the `in` operator; for mappings, `in` should search the mapping's keys; for sequences, it should search through the values. It is further recommended that both mappings and sequences implement the *__iter__()* method to allow efficient iteration through the container; for mappings, *__iter__()* should be the same as `keys()`; for sequences, it should iterate through the values.

`object.`**`__len__`**`(`*self*`)`
: Chamado para implementar a função embutida `len()`. Deve retornar o comprimento do objeto, um inteiro >= 0. Além disso, um objeto que não define um método *__bool__()* e cujo método *__len__()* retorna zero é considerado como falso em um contexto booleano.

  CPython implementation detail: No CPython, o comprimento deve ser no máximo `sys.maxsize`. Se o comprimento for maior que `sys.maxsize`, alguns recursos (como `len()`) podem levantar `OverflowError`. Para evitar levantar `OverflowError` pelo teste de valor de verdade, um objeto deve definir um método *__bool__()*.

`object.`**`__length_hint__`**`(`*self*`)`
: Called to implement `operator.length_hint()`. Should return an estimated length for the object (which may be greater or less than the actual length). The length must be an integer >= 0. This method is purely an optimization and is never required for correctness.

  Novo na versão 3.4.

Nota: O fatiamento é feito exclusivamente com os três métodos a seguir. Uma chamada como

```
a[1:2] = b
```

é traduzida com

```
a[slice(1, 2, None)] = b
```

e assim por diante. Os itens de fatia ausentes são sempre preenchidos com `None`.

`object.`**`__getitem__`**`(`*self*`, `*key*`)`
: Chamado para implementar a avaliação de `self[key]`. Para tipos de sequência, as chaves aceitas devem ser inteiros e objetos fatia. Observe que a interpretação especial de índices negativos (se a classe deseja emular um tipo de sequência) depende do método *__getitem__()*. Se *key* for de um tipo impróprio, `TypeError` pode ser levantada; se de um valor fora do conjunto de índices para a sequência (após qualquer interpretação especial de valores negativos), `IndexError` deve ser levantada. Para tipos de mapeamento, se *key* estiver faltando (não no contêiner), `KeyError` deve ser levantada.

Nota: Os loops *for* esperam que uma `IndexError` seja levantada para índices ilegais para permitir a detecção apropriada do fim da sequência.

`object.`**`__setitem__`**(*self*, *key*, *value*)
Chamado para implementar a atribuição de `self[key]`. Mesma nota que para *__getitem__()*. Isso só deve ser implementado para mapeamentos se os objetos suportarem alterações nos valores das chaves, ou se novas chaves puderem ser adicionadas, ou para sequências se os elementos puderem ser substituídos. As mesmas exceções devem ser levantadas para valores *key* impróprios do método *__getitem__()*.

`object.`**`__delitem__`**(*self*, *key*)
Chamado para implementar a exclusão de `self[key]`. Mesma nota que para *__getitem__()*. Isso só deve ser implementado para mapeamentos se os objetos suportarem remoções de chaves, ou para sequências se os elementos puderem ser removidos da sequência. As mesmas exceções devem ser levantadas para valores *key* impróprios do método *__getitem__()*.

`object.`**`__missing__`**(*self*, *key*)
Chamado por `dict.`*__getitem__()* para implementar `self[key]` para subclasses de dicionário quando a chave não estiver no dicionário.

`object.`**`__iter__`**(*self*)
Este método é chamado quando um iterador é necessário para um contêiner. Este método deve retornar um novo objeto iterador que pode iterar sobre todos os objetos no contêiner. Para mapeamentos, ele deve iterar sobre as chaves do contêiner.

Os objetos iteradores também precisam implementar este método; eles são obrigados a retornar. Para obter mais informações sobre objetos iteradores, consulte typeiter.

`object.`**`__reversed__`**(*self*)
Chamado (se presente) pelo `reversed()` embutido para implementar a iteração reversa. Ele deve retornar um novo objeto iterador que itera sobre todos os objetos no contêiner na ordem reversa.

Se o método *__reversed__()* não for fornecido, o `reversed()` embutido voltará a usar o protocolo de sequência (*__len__()* e *__getitem__()*). Objetos que suportam o protocolo de sequência só devem fornecer *__reversed__()* se eles puderem fornecer uma implementação que seja mais eficiente do que aquela fornecida por `reversed()`.

The membership test operators (*in* and *not in*) are normally implemented as an iteration through a sequence. However, container objects can supply the following special method with a more efficient implementation, which also does not require the object be a sequence.

`object.`**`__contains__`**(*self*, *item*)
Chamado para implementar operadores de teste de associação. Deve retornar verdadeiro se *item* estiver em *self*, falso caso contrário. Para objetos de mapeamento, isso deve considerar as chaves do mapeamento em vez dos valores ou pares de itens-chave.

Para objetos que não definem *__contains__()*, o teste de associação primeiro tenta a iteração via *__iter__()*, depois o protocolo de iteração de sequência antigo via *__getitem__()*, consulte *esta seção em a referência da linguagem*.

3.3.7 Emulando tipos numéricos

Os métodos a seguir podem ser definidos para emular objetos numéricos. Métodos correspondentes a operações que não são suportadas pelo tipo particular de número implementado (por exemplo, operações bit a bit para números não inteiros) devem ser deixados indefinidos.

`object.`**`__add__`**(*self*, *other*)
`object.`**`__sub__`**(*self*, *other*)
`object.`**`__mul__`**(*self*, *other*)
`object.`**`__matmul__`**(*self*, *other*)
`object.`**`__truediv__`**(*self*, *other*)
`object.`**`__floordiv__`**(*self*, *other*)
`object.`**`__mod__`**(*self*, *other*)
`object.`**`__divmod__`**(*self*, *other*)
`object.`**`__pow__`**(*self*, *other*[, *modulo*])
`object.`**`__lshift__`**(*self*, *other*)
`object.`**`__rshift__`**(*self*, *other*)
`object.`**`__and__`**(*self*, *other*)
`object.`**`__xor__`**(*self*, *other*)
`object.`**`__or__`**(*self*, *other*)

Esses métodos são chamados para implementar as operações aritméticas binárias (`+`, `-`, `*`, `@`, `/`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`). Por exemplo, para avaliar a expressão `x + y`, onde *x* é uma instância de uma classe que tem um método *__add__()*, `x.__add__(y)` é chamado. O método *__divmod__()* deve ser equivalente a usar *__floordiv__()* e *__mod__()*; não deve estar relacionado a *__truediv__()*. Note que *__pow__()* deve ser definido para aceitar um terceiro argumento opcional se a versão ternária da função interna `pow()` for suportada.

Se um desses métodos não suporta a operação com os argumentos fornecidos, ele deve retornar `NotImplemented`.

`object.`**`__radd__`**(*self*, *other*)
`object.`**`__rsub__`**(*self*, *other*)
`object.`**`__rmul__`**(*self*, *other*)
`object.`**`__rmatmul__`**(*self*, *other*)
`object.`**`__rtruediv__`**(*self*, *other*)
`object.`**`__rfloordiv__`**(*self*, *other*)
`object.`**`__rmod__`**(*self*, *other*)
`object.`**`__rdivmod__`**(*self*, *other*)
`object.`**`__rpow__`**(*self*, *other*)
`object.`**`__rlshift__`**(*self*, *other*)
`object.`**`__rrshift__`**(*self*, *other*)
`object.`**`__rand__`**(*self*, *other*)
`object.`**`__rxor__`**(*self*, *other*)
`object.`**`__ror__`**(*self*, *other*)

Esses métodos são chamados para implementar as operações aritméticas binárias (`+`, `-`, `*`, `@`, `/`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`) com operandos refletidos (trocados). Essas funções são chamadas apenas se o operando esquerdo não suportar a operação correspondente[3] e os operandos forem de tipos diferentes.[4] Por exemplo, para avaliar a expressão `x - y`, onde *y* é uma instância de uma classe que tem um método *__rsub__()*, `y.__rsub__(x)` é chamado se `x.__sub__(y)` retorna *NotImplemented*.

Note que ternário `pow()` não tentará chamar *__rpow__()* (as regras de coerção se tornariam muito complicadas).

[3] "Não suportar" aqui significa que a classe não possui tal método, ou o método retorna `NotImplemented`. Não defina o método como `None` se quiser forçar o fallback para o método refletido do operando correto – isso terá o efeito oposto de *bloquear* explicitamente esse fallback.

[4] For operands of the same type, it is assumed that if the non-reflected method (such as *__add__()*) fails the operation is not supported, which is why the reflected method is not called.

Nota: If the right operand's type is a subclass of the left operand's type and that subclass provides the reflected method for the operation, this method will be called before the left operand's non-reflected method. This behavior allows subclasses to override their ancestors' operations.

object.**__iadd__**(*self*, *other*)
object.**__isub__**(*self*, *other*)
object.**__imul__**(*self*, *other*)
object.**__imatmul__**(*self*, *other*)
object.**__itruediv__**(*self*, *other*)
object.**__ifloordiv__**(*self*, *other*)
object.**__imod__**(*self*, *other*)
object.**__ipow__**(*self*, *other*[, *modulo*])
object.**__ilshift__**(*self*, *other*)
object.**__irshift__**(*self*, *other*)
object.**__iand__**(*self*, *other*)
object.**__ixor__**(*self*, *other*)
object.**__ior__**(*self*, *other*)

Esses métodos são chamados para implementar as atribuições aritméticas aumentadas (+=, -=, *=, @=, /=, //=, %=, **=, <<=, >>=, &=, ^=, |=). Esses métodos devem tentar fazer a operação no local (modificando *self*) e retornar o resultado (que poderia ser, mas não precisa ser, *self*). Se um método específico não for definido, a atribuição aumentada volta aos métodos normais. Por exemplo, se *x* é uma instância de uma classe com um método *__iadd__()*, `x += y` é equivalente a `x = x.__iadd__(y)`. Caso contrário, `x.__add__(y)` e `y.__radd__(x)` são considerados, como com a avaliação de `x + y`. Em certas situações, a atribuição aumentada pode resultar em erros inesperados (ver faq-augmented-assignment-tuple-error), mas este comportamento é na verdade parte do modelo de dados.

object.**__neg__**(*self*)
object.**__pos__**(*self*)
object.**__abs__**(*self*)
object.**__invert__**(*self*)

Chamado para implementar as operações aritméticas unárias (-, +, `abs()` e ~).

object.**__complex__**(*self*)
object.**__int__**(*self*)
object.**__float__**(*self*)

Chamado para implementar as funções embutidas `complex()`, `int()` e `float()`. Deve retornar um valor do tipo apropriado.

object.**__index__**(*self*)

Chamado para implementar `operator.index()`, e sempre que o Python precisar converter sem perdas o objeto numérico em um objeto inteiro (como no fatiamento ou nas funções embutidas `bin()`, `hex()` e `oct()`). A presença deste método indica que o objeto numérico é do tipo inteiro. Deve retornar um número inteiro.

Nota: In order to have a coherent integer type class, when *__index__()* is defined *__int__()* should also be defined, and both should return the same value.

object.**__round__**(*self*[, *ndigits*])
object.**__trunc__**(*self*)
object.**__floor__**(*self*)
object.**__ceil__**(*self*)

Chamado para implementar as funções embutidas `round()` e `trunc()`, `floor()` e `ceil()` de `math`. A menos que *ndigits* sejam passados para `__round__()` todos estes métodos devem retornar o valor do objeto truncado para um `Integral` (tipicamente um `int`).

Se *__int__()* não for definido, então a função embutida `int()` retrocede para *__trunc__()*.

3.3.8 Com gerenciadores de contexto de instruções

A *context manager* is an object that defines the runtime context to be established when executing a *with* statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. Context managers are normally invoked using the *with* statement (described in section *The with statement*), but can also be used by directly invoking their methods.

Os usos típicos de gerenciadores de contexto incluem salvar e restaurar vários tipos de estado global, bloquear e desbloquear recursos, fechar arquivos abertos, etc.

Para obter mais informações sobre gerenciadores de contexto, consulte typecontextmanager.

`object.`**`__enter__`**`(`*self*`)`
: Enter the runtime context related to this object. The *with* statement will bind this method's return value to the target(s) specified in the *as* clause of the statement, if any.

`object.`**`__exit__`**`(`*self*, *exc_type*, *exc_value*, *traceback*`)`
: Sai do contexto de tempo de execução relacionado a este objeto. Os parâmetros descrevem a exceção que fez com que o contexto fosse encerrado. Se o contexto foi encerrado sem exceção, todos os três argumentos serão `None`.

  Se uma exceção for fornecida e o método desejar suprimir a exceção (ou seja, evitar que ela seja propagada), ele deve retornar um valor verdadeiro. Caso contrário, a exceção será processada normalmente ao sair deste método.

  Observe que os métodos *__exit__()* não devem relançar a exceção passada; esta é a responsabilidade do chamador.

Ver também:

PEP 343 - The "with" statement A especificação, o histórico e os exemplos para a instrução Python *with*.

3.3.9 Pesquisa de método especial

Para classes personalizadas, as invocações implícitas de métodos especiais só têm garantia de funcionar corretamente se definidas em um tipo de objeto, não no dicionário de instância do objeto. Esse comportamento é o motivo pelo qual o código a seguir levanta uma exceção:

```
>>> class C:
...     pass
...
>>> c = C()
>>> c.__len__ = lambda: 5
>>> len(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'C' has no len()
```

A justificativa por trás desse comportamento está em uma série de métodos especiais como *__hash__()* e *__repr__()* que são implementados por todos os objetos, incluindo objetos de tipo. Se a pesquisa implícita desses métodos usasse o processo de pesquisa convencional, eles falhariam quando chamados no próprio objeto do tipo:

```
>>> 1 .__hash__() == hash(1)
True
>>> int.__hash__() == hash(int)
Traceback (most recent call last):
```

(continua na próxima página)

(continuação da página anterior)

```
  File "<stdin>", line 1, in <module>
TypeError: descriptor '__hash__' of 'int' object needs an argument
```

A tentativa incorreta de invocar um método não vinculado de uma classe dessa maneira é às vezes referida como "confusão de metaclasse" e é evitada ignorando a instância ao pesquisar métodos especiais:

```
>>> type(1).__hash__(1) == hash(1)
True
>>> type(int).__hash__(int) == hash(int)
True
```

Além de contornar quaisquer atributos de instância no interesse da correção, a pesquisa de método especial implícita geralmente também contorna o método *__getattribute__()* mesmo da metaclasse do objeto:

```
>>> class Meta(type):
...     def __getattribute__(*args):
...         print("Metaclass getattribute invoked")
...         return type.__getattribute__(*args)
...
>>> class C(object, metaclass=Meta):
...     def __len__(self):
...         return 10
...     def __getattribute__(*args):
...         print("Class getattribute invoked")
...         return object.__getattribute__(*args)
...
>>> c = C()
>>> c.__len__()                 # Explicit lookup via instance
Class getattribute invoked
10
>>> type(c).__len__(c)          # Explicit lookup via type
Metaclass getattribute invoked
10
>>> len(c)                      # Implicit lookup
10
```

Ignorar a maquinaria de *__getattribute__()* desta forma fornece um escopo significativo para otimizações de velocidade dentro do interpretador, ao custo de alguma flexibilidade no tratamento de métodos especiais (o método especial *deve* ser definido no próprio objeto classe em ordem ser invocado de forma consistente pelo interpretador).

3.4 Coroutines

3.4.1 Objetos aguardáveis

An *awaitable* object generally implements an *__await__()* method. *Coroutine* objects returned from *async def* functions are awaitable.

Nota: Os objetos *iteradores geradores* retornados de geradores decorados com `types.coroutine()` ou `asyncio.coroutine()` também são aguardáveis, mas eles não implementam *__await__()*.

`object.`**`__await__`**`(`*self*`)`

Deve retornar um iterador. Deve ser usado para implementar objetos *aguardáveis*. Por exemplo, `asyncio.Future` implementa este método para ser compatível com a expressão `await`.

Novo na versão 3.5.

Ver também:

PEP 492 para informações adicionais sobre objetos aguardáveis.

3.4.2 Objetos Coroutine

Coroutine objects are *awaitable* objects. A coroutine's execution can be controlled by calling `__await__()` and iterating over the result. When the coroutine has finished executing and returns, the iterator raises `StopIteration`, and the exception's `value` attribute holds the return value. If the coroutine raises an exception, it is propagated by the iterator. Coroutines should not directly raise unhandled `StopIteration` exceptions.

As corrotinas também têm os métodos listados abaixo, que são análogos aos dos geradores (ver *Generator-iterator methods*). No entanto, ao contrário dos geradores, as corrotinas não suportam diretamente a iteração.

Alterado na versão 3.5.2: É uma `RuntimeError` para aguardar uma corrotina mais de uma vez.

`coroutine.`**`send`**`(`*value*`)`
Inicia ou retoma a execução da corrotina. Se *value* for `None`, isso é equivalente a avançar o iterador retornado por `__await__()`. Se *value* não for `None`, este método delega para o método `send()` do iterador que causou a suspensão da corrotina. O resultado (valor de retorno, `StopIteration` ou outra exceção) é o mesmo de iterar sobre o valor de retorno `__await__()`, descrito acima.

`coroutine.`**`throw`**`(`*type*`[`, *value*`[`, *traceback*`]]`)
Levanta a exceção especificada na corrotina. Este método delega ao método `throw()` do iterador que causou a suspensão da corrotina, se ela tiver tal método. Caso contrário, a exceção é levantada no ponto de suspensão. O resultado (valor de retorno, `StopIteration` ou outra exceção) é o mesmo de iterar sobre o valor de retorno `__await__()`, descrito acima. Se a exceção não for capturada na corrotina, ela se propagará de volta para o chamador.

`coroutine.`**`close`**`()`
Faz com que a corrotina se limpe e saia. Se a corrotina for suspensa, este método primeiro delega para o método `close()` do iterador que causou a suspensão da corrotina, se tiver tal método. Então ele levanta `GeneratorExit` no ponto de suspensão, fazendo com que a corrotina se limpe imediatamente. Por fim, a corrotina é marcada como tendo sua execução concluída, mesmo que nunca tenha sido iniciada.

Objetos corrotina são fechados automaticamente usando o processo acima quando estão prestes a ser destruídos.

3.4.3 Iteradores Assíncronos

An *asynchronous iterable* is able to call asynchronous code in its `__aiter__` implementation, and an *asynchronous iterator* can call asynchronous code in its `__anext__` method.

Os iteradores assíncronos podem ser usados em uma instrução `async for`.

`object.`**`__aiter__`**`(`*self*`)`
Deve retornar um objeto *iterador assíncrono*.

`object.`**`__anext__`**`(`*self*`)`
Deve retornar um *aguardável* resultando em um próximo valor do iterador. Deve levantar um erro *exc:'StopAsyncIteration* quando a iteração terminar.

Um exemplo de objeto iterável assíncrono:

```
class Reader:
    async def readline(self):
        ...

    def __aiter__(self):
        return self

    async def __anext__(self):
        val = await self.readline()
        if val == b'':
            raise StopAsyncIteration
        return val
```

Novo na versão 3.5.

Nota: Alterado na versão 3.5.2: Starting with CPython 3.5.2, `__aiter__` can directly return *asynchronous iterators*. Returning an *awaitable* object will result in a `PendingDeprecationWarning`.

The recommended way of writing backwards compatible code in CPython 3.5.x is to continue returning awaitables from `__aiter__`. If you want to avoid the PendingDeprecationWarning and keep the code backwards compatible, the following decorator can be used:

```
import functools
import sys

if sys.version_info < (3, 5, 2):
    def aiter_compat(func):
        @functools.wraps(func)
        async def wrapper(self):
            return func(self)
        return wrapper
else:
    def aiter_compat(func):
        return func
```

Exemplo:

```
class AsyncIterator:

    @aiter_compat
    def __aiter__(self):
        return self

    async def __anext__(self):
        ...
```

Starting with CPython 3.6, the `PendingDeprecationWarning` will be replaced with the `DeprecationWarning`. In CPython 3.7, returning an awaitable from `__aiter__` will result in a `RuntimeError`.

3.4.4 Gerenciadores de contexto assíncronos

Um *gerenciador de contexto assíncrono* é um *gerenciador de contexto* que é capaz de suspender a execução em seus métodos `__aenter__` e `__aexit__`.

Os gerenciadores de contexto assíncronos podem ser usados em uma instrução *async with*.

`object.`**`__aenter__`**`(`*self*`)`
This method is semantically similar to the *__enter__()*, with only difference that it must return an *awaitable*.

`object.`**`__aexit__`**`(`*self*, *exc_type*, *exc_value*, *traceback*`)`
This method is semantically similar to the *__exit__()*, with only difference that it must return an *awaitable*.

Um exemplo de uma classe gerenciadora de contexto assíncrona:

```
class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')

    async def __aexit__(self, exc_type, exc, tb):
        await log('exiting context')
```

Novo na versão 3.5.

Notas de Rodapé

CAPÍTULO 4

Modelo de Execussão

4.1 Estrutura de um programa

Um programa Python é construído a partir de blocos de código. Um *bloco* é um pedaço do texto do programa Python que é executado como uma unidade. A seguir estão os blocos: um módulo, um corpo de função e uma definição de classe. Cada comando digitado interativamente é um bloco. Um arquivo de script (um arquivo fornecido como entrada padrão para o interpretador ou especificado como argumento de linha de comando para o interpretador) é um bloco de código. Um comando de script (um comando especificado na linha de comando do interpretador com a opção `-c`) é um bloco de código. O argumento da string passado para as funções embutidas `eval()` e `exec()` é um bloco de código.

Um bloco de código é executado em um *quadro de execução*. Um quadro contém algumas informações administrativas (usadas para depuração) e determina onde e como a execução continua após a conclusão do bloco de código.

4.2 Nomeação e ligação

4.2.1 Ligação de nomes

Nomes referem-se a objetos. Os nomes são introduzidos por operações de ligação de nomes.

The following constructs bind names: formal parameters to functions, *import* statements, class and function definitions (these bind the class or function name in the defining block), and targets that are identifiers if occurring in an assignment, *for* loop header, or after *as* in a *with* statement or *except* clause. The *import* statement of the form `from ... import *` binds all names defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.

Um alvo ocorrendo em uma instrução *del* também é considerado ligado a esse propósito (embora a semântica real seja para desligar do nome).

Cada atribuição ou instrução de importação ocorre dentro de um bloco definido por uma definição de classe ou função ou no nível do módulo (o bloco de código de nível superior).

Se um nome está ligado a um bloco, é uma variável local desse bloco, a menos que declarado como *nonlocal* ou *global*. Se um nome está ligado a nível do módulo, é uma variável global. (As variáveis do bloco de código do módulo são locais e globais.) Se uma variável for usada em um bloco de código, mas não definida lá, é uma *variável livre*.

Cada ocorrência de um nome no texto do programa se refere à *ligação* daquele nome estabelecido pelas seguintes regras de resolução de nome.

4.2.2 Resolução de nomes

O *escopo* define a visibilidade de um nome dentro de um bloco. Se uma variável local é definida em um bloco, seu escopo inclui esse bloco. Se a definição ocorrer em um bloco de função, o escopo se estende a quaisquer blocos contidos no bloco de definição, a menos que um bloco contido introduza uma ligação diferente para o nome.

Quando um nome é usado em um bloco de código, ele é resolvido usando o escopo envolvente mais próximo. O conjunto de todos esses escopos visíveis a um bloco de código é chamado de *ambiente* do bloco.

Quando um nome não é encontrado, uma exceção `NameError` é levantada. Se o escopo atual for um escopo de função e o nome se referir a uma variável local que ainda não foi associada a um valor no ponto onde o nome é usado, uma exceção `UnboundLocalError` é levantada. `UnboundLocalError` é uma subclasse de `NameError`.

If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block. This can lead to errors when a name is used within a block before it is bound. This rule is subtle. Python lacks declarations and allows name binding operations to occur anywhere within a code block. The local variables of a code block can be determined by scanning the entire text of the block for name binding operations.

If the *global* statement occurs within a block, all uses of the name specified in the statement refer to the binding of that name in the top-level namespace. Names are resolved in the top-level namespace by searching the global namespace, i.e. the namespace of the module containing the code block, and the builtins namespace, the namespace of the module `builtins`. The global namespace is searched first. If the name is not found there, the builtins namespace is searched. The *global* statement must precede all uses of the name.

The *global* statement has the same scope as a name binding operation in the same block. If the nearest enclosing scope for a free variable contains a global statement, the free variable is treated as a global.

The *nonlocal* statement causes corresponding names to refer to previously bound variables in the nearest enclosing function scope. `SyntaxError` is raised at compile time if the given name does not exist in any enclosing function scope.

The namespace for a module is automatically created the first time a module is imported. The main module for a script is always called `__main__`.

Class definition blocks and arguments to `exec()` and `eval()` are special in the context of name resolution. A class definition is an executable statement that may use and define names. These references follow the normal rules for name resolution with an exception that unbound local variables are looked up in the global namespace. The namespace of the class definition becomes the attribute dictionary of the class. The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods – this includes comprehensions and generator expressions since they are implemented using a function scope. This means that the following will fail:

```
class A:
    a = 42
    b = list(a + i for i in range(10))
```

4.2.3 Builtins and restricted execution

CPython implementation detail: Users should not touch `__builtins__`; it is strictly an implementation detail. Users wanting to override values in the builtins namespace should *import* the `builtins` module and modify its attributes appropriately.

The builtins namespace associated with the execution of a code block is actually found by looking up the name `__builtins__` in its global namespace; this should be a dictionary or a module (in the latter case the module's dictionary is used). By default, when in the `__main__` module, `__builtins__` is the built-in module `builtins`; when in any other module, `__builtins__` is an alias for the dictionary of the `builtins` module itself.

4.2.4 Interaction with dynamic features

Name resolution of free variables occurs at runtime, not at compile time. This means that the following code will print 42:

```
i = 10
def f():
    print(i)
i = 42
f()
```

The `eval()` and `exec()` functions do not have access to the full environment for resolving names. Names may be resolved in the local and global namespaces of the caller. Free variables are not resolved in the nearest enclosing namespace, but in the global namespace.[1] The `exec()` and `eval()` functions have optional arguments to override the global and local namespace. If only one namespace is specified, it is used for both.

4.3 Exceções

Exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions. An exception is *raised* at the point where the error is detected; it may be *handled* by the surrounding code block or by any code block that directly or indirectly invoked the code block where the error occurred.

The Python interpreter raises an exception when it detects a run-time error (such as division by zero). A Python program can also explicitly raise an exception with the *raise* statement. Exception handlers are specified with the *try* ... *except* statement. The *finally* clause of such a statement can be used to specify cleanup code which does not handle the exception, but is executed whether an exception occurred or not in the preceding code.

Python uses the "termination" model of error handling: an exception handler can find out what happened and continue execution at an outer level, but it cannot repair the cause of the error and retry the failing operation (except by re-entering the offending piece of code from the top).

When an exception is not handled at all, the interpreter terminates execution of the program, or returns to its interactive main loop. In either case, it prints a stack backtrace, except when the exception is `SystemExit`.

Exceptions are identified by class instances. The *except* clause is selected depending on the class of the instance: it must reference the class of the instance or a base class thereof. The instance can be received by the handler and can carry additional information about the exceptional condition.

Nota: Exception messages are not part of the Python API. Their contents may change from one version of Python to the next without warning and should not be relied on by code which will run under multiple versions of the interpreter.

---

[1] This limitation occurs because the code that is executed by these operations is not available at the time the module is compiled.

See also the description of the `try` statement in section *The try statement* and `raise` statement in section *The raise statement*.

Notas de Rodapé

CAPÍTULO 5

O sistema de importação

O código Python em um *módulo* obtém acesso ao código em outro módulo pelo processo de *importação* dele. A instrução `import` é a maneira mais comum de chamar o mecanismo de importação, mas não é a única maneira. Funções como `importlib.import_module()` e a embutida `__import__()` também podem ser usadas para chamar o mecanismo de importação.

The `import` statement combines two operations; it searches for the named module, then it binds the results of that search to a name in the local scope. The search operation of the `import` statement is defined as a call to the `__import__()` function, with the appropriate arguments. The return value of `__import__()` is used to perform the name binding operation of the `import` statement. See the `import` statement for the exact details of that name binding operation.

Uma chamada direta para `__import__()` realiza apenas a pesquisa do módulo e, se encontrada, a operação de criação do módulo. Embora certos efeitos colaterais possam ocorrer, como a importação de pacotes pai e a atualização de vários caches (incluindo `sys.modules`), apenas a instrução `import` realiza uma operação de ligação de nome.

When calling `__import__()` as part of an import statement, the standard builtin `__import__()` is called. Other mechanisms for invoking the import system (such as `importlib.import_module()`) may choose to subvert `__import__()` and use its own solution to implement import semantics.

Quando um módulo é importado pela primeira vez, o Python procura pelo módulo e, se encontrado, cria um objeto de módulo[1], inicializando-o. Se o módulo nomeado não puder ser encontrado, uma `ModuleNotFoundError` será levantada. O Python implementa várias estratégias para procurar o módulo nomeado quando o mecanismo de importação é chamado. Essas estratégias podem ser modificadas e estendidas usando vários ganchos descritos nas seções abaixo.

Alterado na versão 3.3: O sistema de importação foi atualizado para implementar completamente a segunda fase da **PEP 302**. Não há mais um mecanismo de importação implícito – o sistema completo de importação é exposto através de `sys.meta_path`. Além disso, o suporte nativo a pacote de espaço de nomes foi implementado (consulte **PEP 420**).

[1] See `types.ModuleType`.

5.1 importlib

O módulo `importlib` fornece uma API rica para interagir com o sistema de importação. Por exemplo `importlib.import_module()` fornece uma API mais simples e recomendada do que a embutida `__import__()` para chamar o mecanismo de importação. Consulte a documentação da biblioteca `importlib` para obter detalhes adicionais.

5.2 Pacotes

O Python possui apenas um tipo de objeto de módulo e todos os módulos são desse tipo, independentemente de o módulo estar implementado em Python, C ou qualquer outra coisa. Para ajudar a organizar os módulos e fornecer uma hierarquia de nomes, o Python tem o conceito de *pacotes*.

Você pode pensar em pacotes como os diretórios em um sistema de arquivos e os módulos como arquivos nos diretórios, mas não tome essa analogia muito literalmente, já que pacotes e módulos não precisam se originar do sistema de arquivos. Para os fins desta documentação, usaremos essa analogia conveniente de diretórios e arquivos. Como os diretórios do sistema de arquivos, os pacotes são organizados hierarquicamente e os próprios pacotes podem conter subpacotes e módulos regulares.

É importante ter em mente que todos os pacotes são módulos, mas nem todos os módulos são pacotes. Ou, dito de outra forma, os pacotes são apenas um tipo especial de módulo. Especificamente, qualquer módulo que contenha um atributo `__path__` é considerado um pacote.

Todos os módulos têm um nome. Os nomes dos subpacotes são separados do nome do pacote pai por pontos, semelhante à sintaxe de acesso aos atributos padrão do Python. Assim, você pode ter um módulo chamado `sys` e um pacote chamado `email`, que por sua vez possui um subpacote chamado `email.mime` e um módulo dentro desse subpacote chamado `email.mime.text`.

5.2.1 Pacotes regulares

O Python define dois tipos de pacotes, *pacotes regulares* e *pacotes de espaço de nomes*. Pacotes regulares são pacotes tradicionais, como existiam no Python 3.2 e versões anteriores. Um pacote regular é normalmente implementado como um diretório que contém um arquivo `__init__.py`. Quando um pacote regular é importado, esse arquivo `__init__.py` é executado implicitamente, e os objetos que ele define são vinculados aos nomes no espaço de nomes do pacote. O arquivo `__init__.py` pode conter o mesmo código Python que qualquer outro módulo pode conter, e o Python adicionará alguns atributos adicionais ao módulo quando ele for importado.

Por exemplo, o layout do sistema de arquivos a seguir define um pacote `parent` de nível superior com três subpacotes:

```
parent/
    __init__.py
    one/
        __init__.py
    two/
        __init__.py
    three/
        __init__.py
```

A importação de `parent.one` vai executar implicitamente `parent/__init__.py` e `parent/one/__init__.py`. Importações subsequentes de `parent.two` ou `parent.three` vão executar `parent/two/__init__.py` e `parent/three/__init__.py`, respectivamente.

5.2.2 Pacotes de espaço de nomes

Um pacote de espaço de nomes é um composto de várias *porções*, em que cada parte contribui com um subpacote para o pacote pai. Partes podem residir em locais diferentes no sistema de arquivos. Partes também podem ser encontradas em arquivos zip, na rede ou em qualquer outro lugar que o Python pesquisar durante a importação. Os pacotes de espaço de nomes podem ou não corresponder diretamente aos objetos no sistema de arquivos; eles podem ser módulos virtuais que não têm representação concreta.

Os pacotes de espaço de nomes não usam uma lista comum para o atributo `__path__`. Em vez disso, eles usam um tipo iterável personalizado que executará automaticamente uma nova pesquisa por partes do pacote na próxima tentativa de importação dentro desse pacote, se o caminho do pacote pai (ou `sys.path` para um pacote de nível superior) for alterado.

Com pacotes de espaço de nomes, não há arquivo `pai/__init__.py`. De fato, pode haver vários diretórios `pai` encontrados durante a pesquisa de importação, onde cada um é fornecido por uma parte diferente. Portanto, `pai/um` pode não estar fisicamente localizado próximo a `pai/dois`. Nesse caso, o Python criará um pacote de espaço de nomes para o pacote `pai` de nível superior sempre que ele ou um de seus subpacotes for importado.

Veja também **PEP 420** para a especificação de pacotes de espaço de nomes.

5.3 Searching

To begin the search, Python needs the *fully qualified* name of the module (or package, but for the purposes of this discussion, the difference is immaterial) being imported. This name may come from various arguments to the *import* statement, or from the parameters to the `importlib.import_module()` or `__import__()` functions.

This name will be used in various phases of the import search, and it may be the dotted path to a submodule, e.g. `foo.bar.baz`. In this case, Python first tries to import `foo`, then `foo.bar`, and finally `foo.bar.baz`. If any of the intermediate imports fail, a `ModuleNotFoundError` is raised.

5.3.1 The module cache

The first place checked during import search is `sys.modules`. This mapping serves as a cache of all modules that have been previously imported, including the intermediate paths. So if `foo.bar.baz` was previously imported, `sys.modules` will contain entries for `foo`, `foo.bar`, and `foo.bar.baz`. Each key will have as its value the corresponding module object.

During import, the module name is looked up in `sys.modules` and if present, the associated value is the module satisfying the import, and the process completes. However, if the value is `None`, then a `ModuleNotFoundError` is raised. If the module name is missing, Python will continue searching for the module.

`sys.modules` is writable. Deleting a key may not destroy the associated module (as other modules may hold references to it), but it will invalidate the cache entry for the named module, causing Python to search anew for the named module upon its next import. The key can also be assigned to `None`, forcing the next import of the module to result in a `ModuleNotFoundError`.

Beware though, as if you keep a reference to the module object, invalidate its cache entry in `sys.modules`, and then re-import the named module, the two module objects will *not* be the same. By contrast, `importlib.reload()` will reuse the *same* module object, and simply reinitialise the module contents by rerunning the module's code.

5.3.2 Finders and loaders

If the named module is not found in `sys.modules`, then Python's import protocol is invoked to find and load the module. This protocol consists of two conceptual objects, *finders* and *loaders*. A finder's job is to determine whether it can find the named module using whatever strategy it knows about. Objects that implement both of these interfaces are referred to as *importers* - they return themselves when they find that they can load the requested module.

Python includes a number of default finders and importers. The first one knows how to locate built-in modules, and the second knows how to locate frozen modules. A third default finder searches an *import path* for modules. The *import path* is a list of locations that may name file system paths or zip files. It can also be extended to search for any locatable resource, such as those identified by URLs.

The import machinery is extensible, so new finders can be added to extend the range and scope of module searching.

Finders do not actually load modules. If they can find the named module, they return a *module spec*, an encapsulation of the module's import-related information, which the import machinery then uses when loading the module.

The following sections describe the protocol for finders and loaders in more detail, including how you can create and register new ones to extend the import machinery.

Alterado na versão 3.4: In previous versions of Python, finders returned *loaders* directly, whereas now they return module specs which *contain* loaders. Loaders are still used during import but have fewer responsibilities.

5.3.3 Import hooks

The import machinery is designed to be extensible; the primary mechanism for this are the *import hooks*. There are two types of import hooks: *meta hooks* and *import path hooks*.

Meta hooks are called at the start of import processing, before any other import processing has occurred, other than `sys.modules` cache look up. This allows meta hooks to override `sys.path` processing, frozen modules, or even built-in modules. Meta hooks are registered by adding new finder objects to `sys.meta_path`, as described below.

Import path hooks are called as part of `sys.path` (or `package.__path__`) processing, at the point where their associated path item is encountered. Import path hooks are registered by adding new callables to `sys.path_hooks` as described below.

5.3.4 The meta path

When the named module is not found in `sys.modules`, Python next searches `sys.meta_path`, which contains a list of meta path finder objects. These finders are queried in order to see if they know how to handle the named module. Meta path finders must implement a method called `find_spec()` which takes three arguments: a name, an import path, and (optionally) a target module. The meta path finder can use any strategy it wants to determine whether it can handle the named module or not.

If the meta path finder knows how to handle the named module, it returns a spec object. If it cannot handle the named module, it returns `None`. If `sys.meta_path` processing reaches the end of its list without returning a spec, then a `ModuleNotFoundError` is raised. Any other exceptions raised are simply propagated up, aborting the import process.

The `find_spec()` method of meta path finders is called with two or three arguments. The first is the fully qualified name of the module being imported, for example `foo.bar.baz`. The second argument is the path entries to use for the module search. For top-level modules, the second argument is `None`, but for submodules or subpackages, the second argument is the value of the parent package's `__path__` attribute. If the appropriate `__path__` attribute cannot be accessed, a `ModuleNotFoundError` is raised. The third argument is an existing module object that will be the target of loading later. The import system passes in a target module only during reload.

The meta path may be traversed multiple times for a single import request. For example, assuming none of the modules involved has already been cached, importing `foo.bar.baz` will first perform a top level import, calling `mpf.find_spec("foo", None, None)` on each meta path finder (`mpf`). After `foo` has been imported, `foo.bar` will be imported by traversing the meta path a second time, calling `mpf.find_spec("foo.bar", foo.__path__, None)`. Once `foo.bar` has been imported, the final traversal will call `mpf.find_spec("foo.bar.baz", foo.bar.__path__, None)`.

Some meta path finders only support top level imports. These importers will always return `None` when anything other than `None` is passed as the second argument.

Python's default `sys.meta_path` has three meta path finders, one that knows how to import built-in modules, one that knows how to import frozen modules, and one that knows how to import modules from an *import path* (i.e. the *path based finder*).

Alterado na versão 3.4: The `find_spec()` method of meta path finders replaced `find_module()`, which is now deprecated. While it will continue to work without change, the import machinery will try it only if the finder does not implement `find_spec()`.

5.4 Loading

If and when a module spec is found, the import machinery will use it (and the loader it contains) when loading the module. Here is an approximation of what happens during the loading portion of import:

```
module = None
if spec.loader is not None and hasattr(spec.loader, 'create_module'):
    # It is assumed 'exec_module' will also be defined on the loader.
    module = spec.loader.create_module(spec)
if module is None:
    module = ModuleType(spec.name)
# The import-related module attributes get set here:
_init_module_attrs(spec, module)

if spec.loader is None:
    if spec.submodule_search_locations is not None:
        # namespace package
        sys.modules[spec.name] = module
    else:
        # unsupported
        raise ImportError
elif not hasattr(spec.loader, 'exec_module'):
    module = spec.loader.load_module(spec.name)
    # Set __loader__ and __package__ if missing.
else:
    sys.modules[spec.name] = module
    try:
        spec.loader.exec_module(module)
    except BaseException:
        try:
            del sys.modules[spec.name]
        except KeyError:
            pass
        raise
return sys.modules[spec.name]
```

Note the following details:

- If there is an existing module object with the given name in `sys.modules`, import will have already returned it.

- The module will exist in `sys.modules` before the loader executes the module code. This is crucial because the module code may (directly or indirectly) import itself; adding it to `sys.modules` beforehand prevents unbounded recursion in the worst case and multiple loading in the best.
- If loading fails, the failing module – and only the failing module – gets removed from `sys.modules`. Any module already in the `sys.modules` cache, and any module that was successfully loaded as a side-effect, must remain in the cache. This contrasts with reloading where even the failing module is left in `sys.modules`.
- After the module is created but before execution, the import machinery sets the import-related module attributes ("_init_module_attrs" in the pseudo-code example above), as summarized in a *later section*.
- Module execution is the key moment of loading in which the module's namespace gets populated. Execution is entirely delegated to the loader, which gets to decide what gets populated and how.
- The module created during loading and passed to exec_module() may not be the one returned at the end of import[2].

Alterado na versão 3.4: The import system has taken over the boilerplate responsibilities of loaders. These were previously performed by the `importlib.abc.Loader.load_module()` method.

5.4.1 Loaders

Module loaders provide the critical function of loading: module execution. The import machinery calls the `importlib.abc.Loader.exec_module()` method with a single argument, the module object to execute. Any value returned from `exec_module()` is ignored.

Loaders must satisfy the following requirements:

- If the module is a Python module (as opposed to a built-in module or a dynamically loaded extension), the loader should execute the module's code in the module's global name space (`module.__dict__`).
- If the loader cannot execute the module, it should raise an `ImportError`, although any other exception raised during `exec_module()` will be propagated.

In many cases, the finder and loader can be the same object; in such cases the `find_spec()` method would just return a spec with the loader set to `self`.

Module loaders may opt in to creating the module object during loading by implementing a `create_module()` method. It takes one argument, the module spec, and returns the new module object to use during loading. `create_module()` does not need to set any attributes on the module object. If the method returns `None`, the import machinery will create the new module itself.

Novo na versão 3.4: The `create_module()` method of loaders.

Alterado na versão 3.4: The `load_module()` method was replaced by `exec_module()` and the import machinery assumed all the boilerplate responsibilities of loading.

For compatibility with existing loaders, the import machinery will use the `load_module()` method of loaders if it exists and the loader does not also implement `exec_module()`. However, `load_module()` has been deprecated and loaders should implement `exec_module()` instead.

The `load_module()` method must implement all the boilerplate loading functionality described above in addition to executing the module. All the same constraints apply, with some additional clarification:

- If there is an existing module object with the given name in `sys.modules`, the loader must use that existing module. (Otherwise, `importlib.reload()` will not work correctly.) If the named module does not exist in `sys.modules`, the loader must create a new module object and add it to `sys.modules`.

[2] The importlib implementation avoids using the return value directly. Instead, it gets the module object by looking the module name up in `sys.modules`. The indirect effect of this is that an imported module may replace itself in `sys.modules`. This is implementation-specific behavior that is not guaranteed to work in other Python implementations.

- The module *must* exist in `sys.modules` before the loader executes the module code, to prevent unbounded recursion or multiple loading.
- If loading fails, the loader must remove any modules it has inserted into `sys.modules`, but it must remove **only** the failing module(s), and only if the loader itself has loaded the module(s) explicitly.

Alterado na versão 3.5: A `DeprecationWarning` is raised when `exec_module()` is defined but `create_module()` is not.

Alterado na versão 3.6: An `ImportError` is raised when `exec_module()` is defined but `create_module()` is not.

5.4.2 Submódulos

When a submodule is loaded using any mechanism (e.g. `importlib` APIs, the `import` or `import-from` statements, or built-in `__import__()`) a binding is placed in the parent module's namespace to the submodule object. For example, if package `spam` has a submodule `foo`, after importing `spam.foo`, `spam` will have an attribute `foo` which is bound to the submodule. Let's say you have the following directory structure:

```
spam/
    __init__.py
    foo.py
    bar.py
```

and `spam/__init__.py` has the following lines in it:

```
from .foo import Foo
from .bar import Bar
```

then executing the following puts a name binding to `foo` and `bar` in the `spam` module:

```
>>> import spam
>>> spam.foo
<module 'spam.foo' from '/tmp/imports/spam/foo.py'>
>>> spam.bar
<module 'spam.bar' from '/tmp/imports/spam/bar.py'>
```

Given Python's familiar name binding rules this might seem surprising, but it's actually a fundamental feature of the import system. The invariant holding is that if you have `sys.modules['spam']` and `sys.modules['spam.foo']` (as you would after the above import), the latter must appear as the `foo` attribute of the former.

5.4.3 Module spec

The import machinery uses a variety of information about each module during import, especially before loading. Most of the information is common to all modules. The purpose of a module's spec is to encapsulate this import-related information on a per-module basis.

Using a spec during import allows state to be transferred between import system components, e.g. between the finder that creates the module spec and the loader that executes it. Most importantly, it allows the import machinery to perform the boilerplate operations of loading, whereas without a module spec the loader had that responsibility.

The module's spec is exposed as the `__spec__` attribute on a module object. See `ModuleSpec` for details on the contents of the module spec.

Novo na versão 3.4.

5.4.4 Import-related module attributes

The import machinery fills in these attributes on each module object during loading, based on the module's spec, before the loader executes the module.

__name__
The __name__ attribute must be set to the fully-qualified name of the module. This name is used to uniquely identify the module in the import system.

__loader__
The __loader__ attribute must be set to the loader object that the import machinery used when loading the module. This is mostly for introspection, but can be used for additional loader-specific functionality, for example getting data associated with a loader.

__package__
The module's __package__ attribute must be set. Its value must be a string, but it can be the same value as its __name__. When the module is a package, its __package__ value should be set to its __name__. When the module is not a package, __package__ should be set to the empty string for top-level modules, or for submodules, to the parent package's name. See **PEP 366** for further details.

This attribute is used instead of __name__ to calculate explicit relative imports for main modules, as defined in **PEP 366**. It is expected to have the same value as __spec__.parent.

Alterado na versão 3.6: The value of __package__ is expected to be the same as __spec__.parent.

__spec__
The __spec__ attribute must be set to the module spec that was used when importing the module. Setting __spec__ appropriately applies equally to *modules initialized during interpreter startup*. The one exception is __main__, where __spec__ is *set to None in some cases*.

When __package__ is not defined, __spec__.parent is used as a fallback.

Novo na versão 3.4.

Alterado na versão 3.6: __spec__.parent is used as a fallback when __package__ is not defined.

__path__
If the module is a package (either regular or namespace), the module object's __path__ attribute must be set. The value must be iterable, but may be empty if __path__ has no further significance. If __path__ is not empty, it must produce strings when iterated over. More details on the semantics of __path__ are given *below*.

Non-package modules should not have a __path__ attribute.

__file__

__cached__
__file__ is optional. If set, this attribute's value must be a string. The import system may opt to leave __file__ unset if it has no semantic meaning (e.g. a module loaded from a database).

If __file__ is set, it may also be appropriate to set the __cached__ attribute which is the path to any compiled version of the code (e.g. byte-compiled file). The file does not need to exist to set this attribute; the path can simply point to where the compiled file would exist (see **PEP 3147**).

It is also appropriate to set __cached__ when __file__ is not set. However, that scenario is quite atypical. Ultimately, the loader is what makes use of __file__ and/or __cached__. So if a loader can load from a cached module but otherwise does not load from a file, that atypical scenario may be appropriate.

5.4.5 module.__path__

By definition, if a module has a __path__ attribute, it is a package.

A package's __path__ attribute is used during imports of its subpackages. Within the import machinery, it functions much the same as sys.path, i.e. providing a list of locations to search for modules during import. However, __path__ is typically much more constrained than sys.path.

__path__ must be an iterable of strings, but it may be empty. The same rules used for sys.path also apply to a package's __path__, and sys.path_hooks (described below) are consulted when traversing a package's __path__.

A package's __init__.py file may set or alter the package's __path__ attribute, and this was typically the way namespace packages were implemented prior to **PEP 420**. With the adoption of **PEP 420**, namespace packages no longer need to supply __init__.py files containing only __path__ manipulation code; the import machinery automatically sets __path__ correctly for the namespace package.

5.4.6 Module reprs

By default, all modules have a usable repr, however depending on the attributes set above, and in the module's spec, you can more explicitly control the repr of module objects.

If the module has a spec (__spec__), the import machinery will try to generate a repr from it. If that fails or there is no spec, the import system will craft a default repr using whatever information is available on the module. It will try to use the module.__name__, module.__file__, and module.__loader__ as input into the repr, with defaults for whatever information is missing.

Here are the exact rules used:

- If the module has a __spec__ attribute, the information in the spec is used to generate the repr. The "name", "loader", "origin", and "has_location" attributes are consulted.
- If the module has a __file__ attribute, this is used as part of the module's repr.
- If the module has no __file__ but does have a __loader__ that is not None, then the loader's repr is used as part of the module's repr.
- Otherwise, just use the module's __name__ in the repr.

Alterado na versão 3.4: Use of loader.module_repr() has been deprecated and the module spec is now used by the import machinery to generate a module repr.

For backward compatibility with Python 3.3, the module repr will be generated by calling the loader's module_repr() method, if defined, before trying either approach described above. However, the method is deprecated.

5.5 The Path Based Finder

As mentioned previously, Python comes with several default meta path finders. One of these, called the *path based finder* (PathFinder), searches an *import path*, which contains a list of *path entries*. Each path entry names a location to search for modules.

The path based finder itself doesn't know how to import anything. Instead, it traverses the individual path entries, associating each of them with a path entry finder that knows how to handle that particular kind of path.

The default set of path entry finders implement all the semantics for finding modules on the file system, handling special file types such as Python source code (.py files), Python byte code (.pyc files) and shared libraries (e.g. .so files). When supported by the zipimport module in the standard library, the default path entry finders also handle loading all of these file types (other than shared libraries) from zipfiles.

Path entries need not be limited to file system locations. They can refer to URLs, database queries, or any other location that can be specified as a string.

The path based finder provides additional hooks and protocols so that you can extend and customize the types of searchable path entries. For example, if you wanted to support path entries as network URLs, you could write a hook that implements HTTP semantics to find modules on the web. This hook (a callable) would return a *path entry finder* supporting the protocol described below, which was then used to get a loader for the module from the web.

A word of warning: this section and the previous both use the term *finder*, distinguishing between them by using the terms *meta path finder* and *path entry finder*. These two types of finders are very similar, support similar protocols, and function in similar ways during the import process, but it's important to keep in mind that they are subtly different. In particular, meta path finders operate at the beginning of the import process, as keyed off the `sys.meta_path` traversal.

By contrast, path entry finders are in a sense an implementation detail of the path based finder, and in fact, if the path based finder were to be removed from `sys.meta_path`, none of the path entry finder semantics would be invoked.

5.5.1 Path entry finders

The *path based finder* is responsible for finding and loading Python modules and packages whose location is specified with a string *path entry*. Most path entries name locations in the file system, but they need not be limited to this.

As a meta path finder, the *path based finder* implements the `find_spec()` protocol previously described, however it exposes additional hooks that can be used to customize how modules are found and loaded from the *import path*.

Three variables are used by the *path based finder*, `sys.path`, `sys.path_hooks` and `sys.path_importer_cache`. The `__path__` attributes on package objects are also used. These provide additional ways that the import machinery can be customized.

`sys.path` contains a list of strings providing search locations for modules and packages. It is initialized from the `PYTHONPATH` environment variable and various other installation- and implementation-specific defaults. Entries in `sys.path` can name directories on the file system, zip files, and potentially other "locations" (see the `site` module) that should be searched for modules, such as URLs, or database queries. Only strings and bytes should be present on `sys.path`; all other data types are ignored. The encoding of bytes entries is determined by the individual *path entry finders*.

The *path based finder* is a *meta path finder*, so the import machinery begins the *import path* search by calling the path based finder's `find_spec()` method as described previously. When the `path` argument to `find_spec()` is given, it will be a list of string paths to traverse - typically a package's `__path__` attribute for an import within that package. If the `path` argument is `None`, this indicates a top level import and `sys.path` is used.

The path based finder iterates over every entry in the search path, and for each of these, looks for an appropriate *path entry finder* (`PathEntryFinder`) for the path entry. Because this can be an expensive operation (e.g. there may be *stat()* call overheads for this search), the path based finder maintains a cache mapping path entries to path entry finders. This cache is maintained in `sys.path_importer_cache` (despite the name, this cache actually stores finder objects rather than being limited to *importer* objects). In this way, the expensive search for a particular *path entry* location's *path entry finder* need only be done once. User code is free to remove cache entries from `sys.path_importer_cache` forcing the path based finder to perform the path entry search again[3].

If the path entry is not present in the cache, the path based finder iterates over every callable in `sys.path_hooks`. Each of the *path entry hooks* in this list is called with a single argument, the path entry to be searched. This callable may either return a *path entry finder* that can handle the path entry, or it may raise `ImportError`. An `ImportError` is used by the path based finder to signal that the hook cannot find a *path entry finder* for that *path entry*. The exception is ignored and *import path* iteration continues. The hook should expect either a string or bytes object; the encoding of bytes objects is up to the hook (e.g. it may be a file system encoding, UTF-8, or something else), and if the hook cannot decode the argument, it should raise `ImportError`.

[3] In legacy code, it is possible to find instances of `imp.NullImporter` in the `sys.path_importer_cache`. It is recommended that code be changed to use `None` instead. See portingpythoncode for more details.

If `sys.path_hooks` iteration ends with no *path entry finder* being returned, then the path based finder's `find_spec()` method will store `None` in `sys.path_importer_cache` (to indicate that there is no finder for this path entry) and return `None`, indicating that this *meta path finder* could not find the module.

If a *path entry finder* *is* returned by one of the *path entry hook* callables on `sys.path_hooks`, then the following protocol is used to ask the finder for a module spec, which is then used when loading the module.

The current working directory – denoted by an empty string – is handled slightly differently from other entries on `sys.path`. First, if the current working directory is found to not exist, no value is stored in `sys.path_importer_cache`. Second, the value for the current working directory is looked up fresh for each module lookup. Third, the path used for `sys.path_importer_cache` and returned by `importlib.machinery.PathFinder.find_spec()` will be the actual current working directory and not the empty string.

5.5.2 Path entry finder protocol

In order to support imports of modules and initialized packages and also to contribute portions to namespace packages, path entry finders must implement the `find_spec()` method.

`find_spec()` takes two argument, the fully qualified name of the module being imported, and the (optional) target module. `find_spec()` returns a fully populated spec for the module. This spec will always have "loader" set (with one exception).

To indicate to the import machinery that the spec represents a namespace *portion*. the path entry finder sets "loader" on the spec to `None` and "submodule_search_locations" to a list containing the portion.

Alterado na versão 3.4: `find_spec()` replaced `find_loader()` and `find_module()`, both of which are now deprecated, but will be used if `find_spec()` is not defined.

Older path entry finders may implement one of these two deprecated methods instead of `find_spec()`. The methods are still respected for the sake of backward compatibility. However, if `find_spec()` is implemented on the path entry finder, the legacy methods are ignored.

`find_loader()` takes one argument, the fully qualified name of the module being imported. `find_loader()` returns a 2-tuple where the first item is the loader and the second item is a namespace *portion*. When the first item (i.e. the loader) is `None`, this means that while the path entry finder does not have a loader for the named module, it knows that the path entry contributes to a namespace portion for the named module. This will almost always be the case where Python is asked to import a namespace package that has no physical presence on the file system. When a path entry finder returns `None` for the loader, the second item of the 2-tuple return value must be a sequence, although it can be empty.

If `find_loader()` returns a non-`None` loader value, the portion is ignored and the loader is returned from the path based finder, terminating the search through the path entries.

For backwards compatibility with other implementations of the import protocol, many path entry finders also support the same, traditional `find_module()` method that meta path finders support. However path entry finder `find_module()` methods are never called with a `path` argument (they are expected to record the appropriate path information from the initial call to the path hook).

The `find_module()` method on path entry finders is deprecated, as it does not allow the path entry finder to contribute portions to namespace packages. If both `find_loader()` and `find_module()` exist on a path entry finder, the import system will always call `find_loader()` in preference to `find_module()`.

5.6 Replacing the standard import system

The most reliable mechanism for replacing the entire import system is to delete the default contents of `sys.meta_path`, replacing them entirely with a custom meta path hook.

If it is acceptable to only alter the behaviour of import statements without affecting other APIs that access the import system, then replacing the builtin `__import__()` function may be sufficient. This technique may also be employed at the module level to only alter the behaviour of import statements within that module.

To selectively prevent import of some modules from a hook early on the meta path (rather than disabling the standard import system entirely), it is sufficient to raise `ModuleNotFoundError` directly from `find_spec()` instead of returning `None`. The latter indicates that the meta path search should continue, while raising an exception terminates it immediately.

5.7 Special considerations for __main__

The `__main__` module is a special case relative to Python's import system. As noted *elsewhere*, the `__main__` module is directly initialized at interpreter startup, much like `sys` and `builtins`. However, unlike those two, it doesn't strictly qualify as a built-in module. This is because the manner in which `__main__` is initialized depends on the flags and other options with which the interpreter is invoked.

5.7.1 __main__.__spec__

Depending on how `__main__` is initialized, `__main__.__spec__` gets set appropriately or to `None`.

When Python is started with the `-m` option, `__spec__` is set to the module spec of the corresponding module or package. `__spec__` is also populated when the `__main__` module is loaded as part of executing a directory, zipfile or other `sys.path` entry.

In the remaining cases `__main__.__spec__` is set to `None`, as the code used to populate the `__main__` does not correspond directly with an importable module:

- interactive prompt
- `-c` option
- running from stdin
- running directly from a source or bytecode file

Note that `__main__.__spec__` is always `None` in the last case, *even if* the file could technically be imported directly as a module instead. Use the `-m` switch if valid module metadata is desired in `__main__`.

Note also that even when `__main__` corresponds with an importable module and `__main__.__spec__` is set accordingly, they're still considered *distinct* modules. This is due to the fact that blocks guarded by `if __name__ == "__main__":` checks only execute when the module is used to populate the `__main__` namespace, and not during normal import.

5.8 Open issues

XXX It would be really nice to have a diagram.

XXX * (import_machinery.rst) how about a section devoted just to the attributes of modules and packages, perhaps expanding upon or supplanting the related entries in the data model reference page?

XXX runpy, pkgutil, et al in the library manual should all get "See Also" links at the top pointing to the new import system section.

XXX Add more explanation regarding the different ways in which `__main__` is initialized?

XXX Add more info on `__main__` quirks/pitfalls (i.e. copy from **PEP 395**).

5.9 Referências

The import machinery has evolved considerably since Python's early days. The original specification for packages is still available to read, although some details have changed since the writing of that document.

The original specification for `sys.meta_path` was **PEP 302**, with subsequent extension in **PEP 420**.

PEP 420 introduced *namespace packages* for Python 3.3. **PEP 420** also introduced the `find_loader()` protocol as an alternative to `find_module()`.

PEP 366 describes the addition of the `__package__` attribute for explicit relative imports in main modules.

PEP 328 introduced absolute and explicit relative imports and initially proposed `__name__` for semantics **PEP 366** would eventually specify for `__package__`.

PEP 338 defines executing modules as scripts.

PEP 451 adds the encapsulation of per-module import state in spec objects. It also off-loads most of the boilerplate responsibilities of loaders back onto the import machinery. These changes allow the deprecation of several APIs in the import system and also addition of new methods to finders and loaders.

Notas de Rodapé

CAPÍTULO 6

Expressões

Este capítulo explica o significado dos elementos das expressões em Python.

Notas de sintaxe: Neste e nos capítulos seguintes, a notação BNF estendida será usada para descrever a sintaxe, não a análise lexical. Quando (uma alternativa de) uma regra de sintaxe tem a forma

```
name  ::=  othername
```

e nenhuma semântica é fornecida, a semântica desta forma de `name` é a mesma que para `othername`.

6.1 Conversões aritméticas

When a description of an arithmetic operator below uses the phrase “the numeric arguments are converted to a common type,” this means that the operator implementation for built-in types works as follows:

- Se um dos argumentos for um número complexo, o outro será convertido em complexo;
- caso contrário, se um dos argumentos for um número de ponto flutuante, o outro será convertido em ponto flutuante;
- caso contrário, ambos devem ser inteiros e nenhuma conversão é necessária.

Algumas regras adicionais se aplicam a certos operadores (por exemplo, uma string como um argumento à esquerda para o operador ‘%’). As extensões devem definir seu próprio comportamento de conversão.

6.2 Átomos

Os átomos são os elementos mais básicos das expressões. Os átomos mais simples são identificadores ou literais. As formas entre parênteses, colchetes ou chaves também são categorizadas sintaticamente como átomos. A sintaxe para átomos é:

```
atom      ::=  identifier | literal | enclosure
enclosure ::=  parenth_form | list_display | dict_display | set_display
               | generator_expression | yield_atom
```

6.2.1 Identificadores (Nomes)

Um identificador que ocorre como um átomo é um nome. Veja a seção *Identificadores e Keywords* para a definição lexical e a seção *Nomeação e ligação* para documentação de nomenclatura e ligação.

Quando o nome está vinculado a um objeto, a avaliação do átomo produz esse objeto. Quando um nome não está vinculado, uma tentativa de avaliá-lo levanta uma exceção `NameError`.

Private name mangling: When an identifier that textually occurs in a class definition begins with two or more underscore characters and does not end in two or more underscores, it is considered a *private name* of that class. Private names are transformed to a longer form before code is generated for them. The transformation inserts the class name, with leading underscores removed and a single underscore inserted, in front of the name. For example, the identifier `__spam` occurring in a class named `Ham` will be transformed to `_Ham__spam`. This transformation is independent of the syntactical context in which the identifier is used. If the transformed name is extremely long (longer than 255 characters), implementation defined truncation may happen. If the class name consists only of underscores, no transformation is done.

6.2.2 Literais

Python supports string and bytes literals and various numeric literals:

```
literal ::=  stringliteral | bytesliteral
             | integer | floatnumber | imagnumber
```

Evaluation of a literal yields an object of the given type (string, bytes, integer, floating point number, complex number) with the given value. The value may be approximated in the case of floating point and imaginary (complex) literals. See section *Literais* for details.

All literals correspond to immutable data types, and hence the object's identity is less important than its value. Multiple evaluations of literals with the same value (either the same occurrence in the program text or a different occurrence) may obtain the same object or a different object with the same value.

6.2.3 Parenthesized forms

A parenthesized form is an optional expression list enclosed in parentheses:

```
parenth_form  ::=  "(" [starred_expression] ")"
```

A parenthesized expression list yields whatever that expression list yields: if the list contains at least one comma, it yields a tuple; otherwise, it yields the single expression that makes up the expression list.

An empty pair of parentheses yields an empty tuple object. Since tuples are immutable, the rules for literals apply (i.e., two occurrences of the empty tuple may or may not yield the same object).

Note that tuples are not formed by the parentheses, but rather by use of the comma operator. The exception is the empty tuple, for which parentheses *are* required — allowing unparenthesized "nothing" in expressions would cause ambiguities and allow common typos to pass uncaught.

6.2.4 Displays for lists, sets and dictionaries

For constructing a list, a set or a dictionary Python provides special syntax called "displays", each of them in two flavors:

- either the container contents are listed explicitly, or
- they are computed via a set of looping and filtering instructions, called a *comprehension*.

Common syntax elements for comprehensions are:

```
comprehension  ::=  expression comp_for
comp_for       ::=  [ASYNC] "for" target_list "in" or_test [comp_iter]
comp_iter      ::=  comp_for | comp_if
comp_if        ::=  "if" expression_nocond [comp_iter]
```

The comprehension consists of a single expression followed by at least one `for` clause and zero or more `for` or `if` clauses. In this case, the elements of the new container are those that would be produced by considering each of the `for` or `if` clauses a block, nesting from left to right, and evaluating the expression to produce an element each time the innermost block is reached.

Note that the comprehension is executed in a separate scope, so names assigned to in the target list don't "leak" into the enclosing scope.

Since Python 3.6, in an `async def` function, an `async for` clause may be used to iterate over a *asynchronous iterator*. A comprehension in an `async def` function may consist of either a `for` or `async for` clause following the leading expression, may contain additional `for` or `async for` clauses, and may also use `await` expressions. If a comprehension contains either `async for` clauses or `await` expressions it is called an *asynchronous comprehension*. An asynchronous comprehension may suspend the execution of the coroutine function in which it appears. See also **PEP 530**.

6.2.5 List displays

A list display is a possibly empty series of expressions enclosed in square brackets:

```
list_display  ::=  "[" [starred_list | comprehension] "]"
```

A list display yields a new list object, the contents being specified by either a list of expressions or a comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and placed into the list object in that order. When a comprehension is supplied, the list is constructed from the elements resulting from the comprehension.

6.2.6 Set displays

A set display is denoted by curly braces and distinguishable from dictionary displays by the lack of colons separating keys and values:

```
set_display  ::=  "{" (starred_list | comprehension) "}"
```

A set display yields a new mutable set object, the contents being specified by either a sequence of expressions or a comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and added to the set object. When a comprehension is supplied, the set is constructed from the elements resulting from the comprehension.

An empty set cannot be constructed with `{}`; this literal constructs an empty dictionary.

6.2.7 Dictionary displays

A dictionary display is a possibly empty series of key/datum pairs enclosed in curly braces:

```
dict_display        ::=  "{" [key_datum_list | dict_comprehension] "}"
key_datum_list      ::=  key_datum ("," key_datum)* [","]
key_datum           ::=  expression ":" expression | "**" or_expr
dict_comprehension  ::=  expression ":" expression comp_for
```

A dictionary display yields a new dictionary object.

If a comma-separated sequence of key/datum pairs is given, they are evaluated from left to right to define the entries of the dictionary: each key object is used as a key into the dictionary to store the corresponding datum. This means that you can specify the same key multiple times in the key/datum list, and the final dictionary's value for that key will be the last one given.

A double asterisk `**` denotes *dictionary unpacking*. Its operand must be a *mapping*. Each mapping item is added to the new dictionary. Later values replace values already set by earlier key/datum pairs and earlier dictionary unpackings.

Novo na versão 3.5: Unpacking into dictionary displays, originally proposed by **PEP 448**.

A dict comprehension, in contrast to list and set comprehensions, needs two expressions separated with a colon followed by the usual "for" and "if" clauses. When the comprehension is run, the resulting key and value elements are inserted in the new dictionary in the order they are produced.

Restrictions on the types of the key values are listed earlier in section *A hierarquia de tipos padrão*. (To summarize, the key type should be *hashable*, which excludes all mutable objects.) Clashes between duplicate keys are not detected; the

last datum (textually rightmost in the display) stored for a given key value prevails.

6.2.8 Generator expressions

A generator expression is a compact generator notation in parentheses:

```
generator_expression  ::=  "(" expression comp_for ")"
```

A generator expression yields a new generator object. Its syntax is the same as for comprehensions, except that it is enclosed in parentheses instead of brackets or curly braces.

Variables used in the generator expression are evaluated lazily when the `__next__()` method is called for the generator object (in the same fashion as normal generators). However, the leftmost `for` clause is immediately evaluated, so that an error produced by it can be seen before any other possible error in the code that handles the generator expression. Subsequent `for` clauses cannot be evaluated immediately since they may depend on the previous `for` loop. For example: `(x*y for x in range(10) for y in bar(x))`.

The parentheses can be omitted on calls with only one argument. See section *Calls* for details.

Since Python 3.6, if the generator appears in an `async def` function, then `async for` clauses and `await` expressions are permitted as with an asynchronous comprehension. If a generator expression contains either `async for` clauses or `await` expressions it is called an *asynchronous generator expression*. An asynchronous generator expression yields a new asynchronous generator object, which is an asynchronous iterator (see *Iteradores Assíncronos*).

6.2.9 Yield expressions

```
yield_atom        ::=  "(" yield_expression ")"
yield_expression  ::=  "yield" [expression_list | "from" expression]
```

The yield expression is used when defining a *generator* function or an *asynchronous generator* function and thus can only be used in the body of a function definition. Using a yield expression in a function's body causes that function to be a generator, and using it in an `async def` function's body causes that coroutine function to be an asynchronous generator. For example:

```
def gen():  # defines a generator function
    yield 123

async def agen(): # defines an asynchronous generator function
    yield 123
```

Generator functions are described below, while asynchronous generator functions are described separately in section *Funções geradoras assíncronas*.

When a generator function is called, it returns an iterator known as a generator. That generator then controls the execution of the generator function. The execution starts when one of the generator's methods is called. At that time, the execution proceeds to the first yield expression, where it is suspended again, returning the value of `expression_list` to the generator's caller. By suspended, we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, the internal evaluation stack, and the state of any exception handling. When the execution is resumed by calling one of the generator's methods, the function can proceed exactly as if the yield expression were just another external call. The value of the yield expression after resuming depends on the method which resumed the execution. If `__next__()` is used (typically via either a `for` or the `next()` builtin) then the result is `None`. Otherwise, if `send()` is used, then the result will be the value passed in to that method.

All of this makes generator functions quite similar to coroutines; they yield multiple times, they have more than one entry

point and their execution can be suspended. The only difference is that a generator function cannot control where the execution should continue after it yields; the control is always transferred to the generator's caller.

Yield expressions are allowed anywhere in a *try* construct. If the generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), the generator-iterator's *close()* method will be called, allowing any pending *finally* clauses to execute.

When `yield from <expr>` is used, it treats the supplied expression as a subiterator. All values produced by that subiterator are passed directly to the caller of the current generator's methods. Any values passed in with *send()* and any exceptions passed in with *throw()* are passed to the underlying iterator if it has the appropriate methods. If this is not the case, then *send()* will raise `AttributeError` or `TypeError`, while *throw()* will just raise the passed in exception immediately.

When the underlying iterator is complete, the `value` attribute of the raised `StopIteration` instance becomes the value of the yield expression. It can be either set explicitly when raising `StopIteration`, or automatically when the sub-iterator is a generator (by returning a value from the sub-generator).

> Alterado na versão 3.3: Added `yield from <expr>` to delegate control flow to a subiterator.

The parentheses may be omitted when the yield expression is the sole expression on the right hand side of an assignment statement.

Ver também:

PEP 255 - Simple Generators The proposal for adding generators and the *yield* statement to Python.

PEP 342 - Coroutines via Enhanced Generators The proposal to enhance the API and syntax of generators, making them usable as simple coroutines.

PEP 380 - Syntax for Delegating to a Subgenerator The proposal to introduce the `yield_from` syntax, making delegation to sub-generators easy.

PEP 525 - Asynchronous Generators The proposal that expanded on **PEP 492** by adding generator capabilities to coroutine functions.

Generator-iterator methods

This subsection describes the methods of a generator iterator. They can be used to control the execution of a generator function.

Note that calling any of the generator methods below when the generator is already executing raises a `ValueError` exception.

`generator.`**`__next__`**`()`
Starts the execution of a generator function or resumes it at the last executed yield expression. When a generator function is resumed with a *__next__()* method, the current yield expression always evaluates to `None`. The execution then continues to the next yield expression, where the generator is suspended again, and the value of the *expression_list* is returned to *__next__()*'s caller. If the generator exits without yielding another value, a `StopIteration` exception is raised.

This method is normally called implicitly, e.g. by a *for* loop, or by the built-in `next()` function.

`generator.`**`send`**`(`*value*`)`
Resumes the execution and "sends" a value into the generator function. The *value* argument becomes the result of the current yield expression. The *send()* method returns the next value yielded by the generator, or raises `StopIteration` if the generator exits without yielding another value. When *send()* is called to start the generator, it must be called with `None` as the argument, because there is no yield expression that could receive the value.

`generator.`**`throw`**`(`*type*`[`, *value*`[`, *traceback*`]]`)
Raises an exception of type `type` at the point where the generator was paused, and returns the next value yielded

by the generator function. If the generator exits without yielding another value, a `StopIteration` exception is raised. If the generator function does not catch the passed-in exception, or raises a different exception, then that exception propagates to the caller.

`generator.close()`
Raises a `GeneratorExit` at the point where the generator function was paused. If the generator function then exits gracefully, is already closed, or raises `GeneratorExit` (by not catching the exception), close returns to its caller. If the generator yields a value, a `RuntimeError` is raised. If the generator raises any other exception, it is propagated to the caller. *`close()`* does nothing if the generator has already exited due to an exception or normal exit.

Exemplos

Here is a simple example that demonstrates the behavior of generators and generator functions:

```
>>> def echo(value=None):
...     print("Execution starts when 'next()' is called for the first time.")
...     try:
...         while True:
...             try:
...                 value = (yield value)
...             except Exception as e:
...                 value = e
...     finally:
...         print("Don't forget to clean up when 'close()' is called.")
...
>>> generator = echo(1)
>>> print(next(generator))
Execution starts when 'next()' is called for the first time.
1
>>> print(next(generator))
None
>>> print(generator.send(2))
2
>>> generator.throw(TypeError, "spam")
TypeError('spam',)
>>> generator.close()
Don't forget to clean up when 'close()' is called.
```

For examples using `yield from`, see pep-380 in "What's New in Python."

Funções geradoras assíncronas

The presence of a yield expression in a function or method defined using *`async def`* further defines the function as a *asynchronous generator* function.

When an asynchronous generator function is called, it returns an asynchronous iterator known as an asynchronous generator object. That object then controls the execution of the generator function. An asynchronous generator object is typically used in an *`async for`* statement in a coroutine function analogously to how a generator object would be used in a *`for`* statement.

Calling one of the asynchronous generator's methods returns an *awaitable* object, and the execution starts when this object is awaited on. At that time, the execution proceeds to the first yield expression, where it is suspended again, returning the value of *`expression_list`* to the awaiting coroutine. As with a generator, suspension means that all local state is retained, including the current bindings of local variables, the instruction pointer, the internal evaluation stack, and the state of any exception handling. When the execution is resumed by awaiting on the next object returned by the asynchronous

generator's methods, the function can proceed exactly as if the yield expression were just another external call. The value of the yield expression after resuming depends on the method which resumed the execution. If `__anext__()` is used then the result is `None`. Otherwise, if `asend()` is used, then the result will be the value passed in to that method.

In an asynchronous generator function, yield expressions are allowed anywhere in a `try` construct. However, if an asynchronous generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), then a yield expression within a `try` construct could result in a failure to execute pending `finally` clauses. In this case, it is the responsibility of the event loop or scheduler running the asynchronous generator to call the asynchronous generator-iterator's `aclose()` method and run the resulting coroutine object, thus allowing any pending `finally` clauses to execute.

To take care of finalization, an event loop should define a *finalizer* function which takes an asynchronous generator-iterator and presumably calls `aclose()` and executes the coroutine. This *finalizer* may be registered by calling `sys.set_asyncgen_hooks()`. When first iterated over, an asynchronous generator-iterator will store the registered *finalizer* to be called upon finalization. For a reference example of a *finalizer* method see the implementation of `asyncio.Loop.shutdown_asyncgens` in Lib/asyncio/base_events.py.

The expression `yield from <expr>` is a syntax error when used in an asynchronous generator function.

Asynchronous generator-iterator methods

This subsection describes the methods of an asynchronous generator iterator, which are used to control the execution of a generator function.

coroutine agen.**__anext__**()
: Returns an awaitable which when run starts to execute the asynchronous generator or resumes it at the last executed yield expression. When an asynchronous generator function is resumed with a `__anext__()` method, the current yield expression always evaluates to `None` in the returned awaitable, which when run will continue to the next yield expression. The value of the *expression_list* of the yield expression is the value of the `StopIteration` exception raised by the completing coroutine. If the asynchronous generator exits without yielding another value, the awaitable instead raises an `StopAsyncIteration` exception, signalling that the asynchronous iteration has completed.

  This method is normally called implicitly by a `async for` loop.

coroutine agen.**asend**(*value*)
: Returns an awaitable which when run resumes the execution of the asynchronous generator. As with the `send()` method for a generator, this "sends" a value into the asynchronous generator function, and the *value* argument becomes the result of the current yield expression. The awaitable returned by the `asend()` method will return the next value yielded by the generator as the value of the raised `StopIteration`, or raises `StopAsyncIteration` if the asynchronous generator exits without yielding another value. When `asend()` is called to start the asynchronous generator, it must be called with `None` as the argument, because there is no yield expression that could receive the value.

coroutine agen.**athrow**(*type*[, *value*[, *traceback*]])
: Returns an awaitable that raises an exception of type `type` at the point where the asynchronous generator was paused, and returns the next value yielded by the generator function as the value of the raised `StopIteration` exception. If the asynchronous generator exits without yielding another value, an `StopAsyncIteration` exception is raised by the awaitable. If the generator function does not catch the passed-in exception, or raises a different exception, then when the awaitable is run that exception propagates to the caller of the awaitable.

coroutine agen.**aclose**()
: Returns an awaitable that when run will throw a `GeneratorExit` into the asynchronous generator function at the point where it was paused. If the asynchronous generator function then exits gracefully, is already closed, or raises `GeneratorExit` (by not catching the exception), then the returned awaitable will raise a `StopIteration` exception. Any further awaitables returned by subsequent calls to the asynchronous generator will raise a

`StopAsyncIteration` exception. If the asynchronous generator yields a value, a `RuntimeError` is raised by the awaitable. If the asynchronous generator raises any other exception, it is propagated to the caller of the awaitable. If the asynchronous generator has already exited due to an exception or normal exit, then further calls to *aclose()* will return an awaitable that does nothing.

6.3 Primaries

Primaries represent the most tightly bound operations of the language. Their syntax is:

```
primary  ::=  atom | attributeref | subscription | slicing | call
```

6.3.1 Attribute references

An attribute reference is a primary followed by a period and a name:

```
attributeref  ::=  primary "." identifier
```

The primary must evaluate to an object of a type that supports attribute references, which most objects do. This object is then asked to produce the attribute whose name is the identifier. This production can be customized by overriding the *__getattr__()* method. If this attribute is not available, the exception `AttributeError` is raised. Otherwise, the type and value of the object produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

6.3.2 Subscriptions

A subscription selects an item of a sequence (string, tuple or list) or mapping (dictionary) object:

```
subscription  ::=  primary "[" expression_list "]"
```

The primary must evaluate to an object that supports subscription (lists or dictionaries for example). User-defined objects can support subscription by defining a *__getitem__()* method.

For built-in objects, there are two types of objects that support subscription:

If the primary is a mapping, the expression list must evaluate to an object whose value is one of the keys of the mapping, and the subscription selects the value in the mapping that corresponds to that key. (The expression list is a tuple except if it has exactly one item.)

If the primary is a sequence, the expression list must evaluate to an integer or a slice (as discussed in the following section).

The formal syntax makes no special provision for negative indices in sequences; however, built-in sequences all provide a *__getitem__()* method that interprets negative indices by adding the length of the sequence to the index (so that `x[-1]` selects the last item of `x`). The resulting value must be a nonnegative integer less than the number of items in the sequence, and the subscription selects the item whose index is that value (counting from zero). Since the support for negative indices and slicing occurs in the object's *__getitem__()* method, subclasses overriding this method will need to explicitly add that support.

A string's items are characters. A character is not a separate data type but a string of exactly one character.

6.3.3 Slicings

A slicing selects a range of items in a sequence object (e.g., a string, tuple or list). Slicings may be used as expressions or as targets in assignment or *del* statements. The syntax for a slicing:

```
slicing      ::=  primary "[" slice_list "]"
slice_list   ::=  slice_item ("," slice_item)* [","]
slice_item   ::=  expression | proper_slice
proper_slice ::=  [lower_bound] ":" [upper_bound] [ ":" [stride] ]
lower_bound  ::=  expression
upper_bound  ::=  expression
stride       ::=  expression
```

There is ambiguity in the formal syntax here: anything that looks like an expression list also looks like a slice list, so any subscription can be interpreted as a slicing. Rather than further complicating the syntax, this is disambiguated by defining that in this case the interpretation as a subscription takes priority over the interpretation as a slicing (this is the case if the slice list contains no proper slice).

The semantics for a slicing are as follows. The primary is indexed (using the same *__getitem__()* method as normal subscription) with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of a proper slice is a slice object (see section *A hierarquia de tipos padrão*) whose `start`, `stop` and `step` attributes are the values of the expressions given as lower bound, upper bound and stride, respectively, substituting `None` for missing expressions.

6.3.4 Calls

A call calls a callable object (e.g., a *function*) with a possibly empty series of *arguments*:

```
call                 ::=  primary "(" [argument_list [","] | comprehension] ")"
argument_list        ::=  positional_arguments ["," starred_and_keywords]
                          ["," keywords_arguments]
                          | starred_and_keywords ["," keywords_arguments]
                          | keywords_arguments
positional_arguments ::=  ["*"] expression ("," ["*"] expression)*
starred_and_keywords ::=  ("*" expression | keyword_item)
                          ("," "*" expression | "," keyword_item)*
keywords_arguments   ::=  (keyword_item | "**" expression)
                          ("," keyword_item | "," "**" expression)*
keyword_item         ::=  identifier "=" expression
```

An optional trailing comma may be present after the positional and keyword arguments but does not affect the semantics.

The primary must evaluate to a callable object (user-defined functions, built-in functions, methods of built-in objects, class objects, methods of class instances, and all objects having a *__call__()* method are callable). All argument expressions are evaluated before the call is attempted. Please refer to section *Definições de função* for the syntax of formal *parameter* lists.

If keyword arguments are present, they are first converted to positional arguments, as follows. First, a list of unfilled slots is created for the formal parameters. If there are N positional arguments, they are placed in the first N slots. Next, for each keyword argument, the identifier is used to determine the corresponding slot (if the identifier is the same as the first formal parameter name, the first slot is used, and so on). If the slot is already filled, a `TypeError` exception is raised.

Otherwise, the value of the argument is placed in the slot, filling it (even if the expression is `None`, it fills the slot). When all arguments have been processed, the slots that are still unfilled are filled with the corresponding default value from the function definition. (Default values are calculated, once, when the function is defined; thus, a mutable object such as a list or dictionary used as default value will be shared by all calls that don't specify an argument value for the corresponding slot; this should usually be avoided.) If there are any unfilled slots for which no default value is specified, a `TypeError` exception is raised. Otherwise, the list of filled slots is used as the argument list for the call.

CPython implementation detail: An implementation may provide built-in functions whose positional parameters do not have names, even if they are 'named' for the purpose of documentation, and which therefore cannot be supplied by keyword. In CPython, this is the case for functions implemented in C that use `PyArg_ParseTuple()` to parse their arguments.

If there are more positional arguments than there are formal parameter slots, a `TypeError` exception is raised, unless a formal parameter using the syntax `*identifier` is present; in this case, that formal parameter receives a tuple containing the excess positional arguments (or an empty tuple if there were no excess positional arguments).

If any keyword argument does not correspond to a formal parameter name, a `TypeError` exception is raised, unless a formal parameter using the syntax `**identifier` is present; in this case, that formal parameter receives a dictionary containing the excess keyword arguments (using the keywords as keys and the argument values as corresponding values), or a (new) empty dictionary if there were no excess keyword arguments.

If the syntax `*expression` appears in the function call, `expression` must evaluate to an *iterable*. Elements from these iterables are treated as if they were additional positional arguments. For the call `f(x1, x2, *y, x3, x4)`, if *y* evaluates to a sequence *y1*, …, *yM*, this is equivalent to a call with M+4 positional arguments *x1*, *x2*, *y1*, …, *yM*, *x3*, *x4*.

A consequence of this is that although the `*expression` syntax may appear *after* explicit keyword arguments, it is processed *before* the keyword arguments (and any `**expression` arguments – see below). So:

```
>>> def f(a, b):
...     print(a, b)
...
>>> f(b=1, *(2,))
2 1
>>> f(a=1, *(2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1, *(2,))
1 2
```

It is unusual for both keyword arguments and the `*expression` syntax to be used in the same call, so in practice this confusion does not arise.

If the syntax `**expression` appears in the function call, `expression` must evaluate to a *mapping*, the contents of which are treated as additional keyword arguments. If a keyword is already present (as an explicit keyword argument, or from another unpacking), a `TypeError` exception is raised.

Formal parameters using the syntax `*identifier` or `**identifier` cannot be used as positional argument slots or as keyword argument names.

Alterado na versão 3.5: Function calls accept any number of `*` and `**` unpackings, positional arguments may follow iterable unpackings (`*`), and keyword arguments may follow dictionary unpackings (`**`). Originally proposed by **PEP 448**.

A call always returns some value, possibly `None`, unless it raises an exception. How this value is computed depends on the type of the callable object.

If it is—

a user-defined function: The code block for the function is executed, passing it the argument list. The first thing the code block will do is bind the formal parameters to the arguments; this is described in section *Definições de função*. When the code block executes a `return` statement, this specifies the return value of the function call.

a built-in function or method: The result is up to the interpreter; see built-in-funcs for the descriptions of built-in functions and methods.

um objeto classe: A new instance of that class is returned.

a class instance method: The corresponding user-defined function is called, with an argument list that is one longer than the argument list of the call: the instance becomes the first argument.

a class instance: The class must define a `__call__()` method; the effect is then the same as if that method was called.

6.4 Await expression

Suspend the execution of *coroutine* on an *awaitable* object. Can only be used inside a *coroutine function*.

```
await_expr  ::=  "await" primary
```

Novo na versão 3.5.

6.5 The power operator

The power operator binds more tightly than unary operators on its left; it binds less tightly than unary operators on its right. The syntax is:

```
power  ::=  (await_expr | primary) ["**" u_expr]
```

Thus, in an unparenthesized sequence of power and unary operators, the operators are evaluated from right to left (this does not constrain the evaluation order for the operands): `-1**2` results in `-1`.

The power operator has the same semantics as the built-in `pow()` function, when called with two arguments: it yields its left argument raised to the power of its right argument. The numeric arguments are first converted to a common type, and the result is of that type.

For int operands, the result has the same type as the operands 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`.

Raising `0.0` to a negative power results in a `ZeroDivisionError`. Raising a negative number to a fractional power results in a `complex` number. (In earlier versions it raised a `ValueError`.)

6.6 Unary arithmetic and bitwise operations

All unary arithmetic and bitwise operations have the same priority:

```
u_expr  ::=  power | "-" u_expr | "+" u_expr | "~" u_expr
```

The unary - (minus) operator yields the negation of its numeric argument.

The unary + (plus) operator yields its numeric argument unchanged.

The unary ~ (invert) operator yields the bitwise inversion of its integer argument. The bitwise inversion of `x` is defined as `-(x+1)`. It only applies to integral numbers.

In all three cases, if the argument does not have the proper type, a `TypeError` exception is raised.

6.7 Binary arithmetic operations

The binary arithmetic operations have the conventional priority levels. Note that some of these operations also apply to certain non-numeric types. Apart from the power operator, there are only two levels, one for multiplicative operators and one for additive operators:

```
m_expr  ::=  u_expr | m_expr "*" u_expr | m_expr "@" m_expr |
             m_expr "//" u_expr | m_expr "/" u_expr |
             m_expr "%" u_expr
a_expr  ::=  m_expr | a_expr "+" m_expr | a_expr "-" m_expr
```

The `*` (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, or one argument must be an integer and the other must be a sequence. In the former case, the numbers are converted to a common type and then multiplied together. In the latter case, sequence repetition is performed; a negative repetition factor yields an empty sequence.

The `@` (at) operator is intended to be used for matrix multiplication. No builtin Python types implement this operator.

Novo na versão 3.5.

The `/` (division) and `//` (floor division) operators yield the quotient of their arguments. The numeric arguments are first converted to a common type. Division of integers yields a float, while floor division of integers results in an integer; the result is that of mathematical division with the 'floor' function applied to the result. Division by zero raises the `ZeroDivisionError` exception.

The `%` (modulo) operator yields the remainder from the division of the first argument by the second. The numeric arguments are first converted to a common type. A zero right argument raises the `ZeroDivisionError` exception. The arguments may be floating point numbers, e.g., `3.14%0.7` equals `0.34` (since `3.14` equals `4*0.7 + 0.34`.) The modulo operator always yields a result with the same sign as its second operand (or zero); the absolute value of the result is strictly smaller than the absolute value of the second operand[1].

The floor division and modulo operators are connected by the following identity: `x == (x//y)*y + (x%y)`. Floor division and modulo are also connected with the built-in function `divmod()`: `divmod(x, y) == (x//y, x%y)`.[2].

[1] While `abs(x%y) < abs(y)` is true mathematically, for floats it may not be true numerically due to roundoff. For example, and assuming a platform on which a Python float is an IEEE 754 double-precision number, in order that `-1e-100 % 1e100` have the same sign as `1e100`, the computed result is `-1e-100 + 1e100`, which is numerically exactly equal to `1e100`. The function `math.fmod()` returns a result whose sign matches the sign of the first argument instead, and so returns `-1e-100` in this case. Which approach is more appropriate depends on the application.

[2] If x is very close to an exact integer multiple of y, it's possible for `x//y` to be one larger than `(x-x%y)//y` due to rounding. In such cases,

In addition to performing the modulo operation on numbers, the `%` operator is also overloaded by string objects to perform old-style string formatting (also known as interpolation). The syntax for string formatting is described in the Python Library Reference, section old-string-formatting.

The floor division operator, the modulo operator, and the `divmod()` function are not defined for complex numbers. Instead, convert to a floating point number using the `abs()` function if appropriate.

The `+` (addition) operator yields the sum of its arguments. The arguments must either both be numbers or both be sequences of the same type. In the former case, the numbers are converted to a common type and then added together. In the latter case, the sequences are concatenated.

The `-` (subtraction) operator yields the difference of its arguments. The numeric arguments are first converted to a common type.

6.8 Shifting operations

The shifting operations have lower priority than the arithmetic operations:

```
shift_expr  ::=  a_expr | shift_expr ("<<" | ">>") a_expr
```

These operators accept integers as arguments. They shift the first argument to the left or right by the number of bits given by the second argument.

A right shift by *n* bits is defined as floor division by `pow(2,n)`. A left shift by *n* bits is defined as multiplication with `pow(2,n)`.

Nota: In the current implementation, the right-hand operand is required to be at most `sys.maxsize`. If the right-hand operand is larger than `sys.maxsize` an `OverflowError` exception is raised.

6.9 Binary bitwise operations

Each of the three bitwise operations has a different priority level:

```
and_expr  ::=  shift_expr | and_expr "&" shift_expr
xor_expr  ::=  and_expr | xor_expr "^" and_expr
or_expr   ::=  xor_expr | or_expr "|" xor_expr
```

The `&` operator yields the bitwise AND of its arguments, which must be integers.

The `^` operator yields the bitwise XOR (exclusive OR) of its arguments, which must be integers.

The `|` operator yields the bitwise (inclusive) OR of its arguments, which must be integers.

Python returns the latter result, in order to preserve that `divmod(x,y)[0] * y + x % y` be very close to `x`.

6.10 Comparações

Unlike C, all comparison operations in Python have the same priority, which is lower than that of any arithmetic, shifting or bitwise operation. Also unlike C, expressions like `a < b < c` have the interpretation that is conventional in mathematics:

```
comparison    ::=  or_expr (comp_operator or_expr)*
comp_operator ::=  "<" | ">" | "==" | ">=" | "<=" | "!="
                   | "is" ["not"] | ["not"] "in"
```

Comparisons yield boolean values: `True` or `False`.

Comparisons can be chained arbitrarily, e.g., `x < y <= z` is equivalent to `x < y and y <= z`, except that `y` is evaluated only once (but in both cases `z` is not evaluated at all when `x < y` is found to be false).

Formally, if *a*, *b*, *c*, …, *y*, *z* are expressions and *op1*, *op2*, …, *opN* are comparison operators, then `a op1 b op2 c ... y opN z` is equivalent to `a op1 b and b op2 c and ... y opN z`, except that each expression is evaluated at most once.

Note that `a op1 b op2 c` doesn't imply any kind of comparison between *a* and *c*, so that, e.g., `x < y > z` is perfectly legal (though perhaps not pretty).

6.10.1 Value comparisons

The operators <, >, ==, >=, <=, and `!=` compare the values of two objects. The objects do not need to have the same type.

Chapter *Objetos, valores e tipos* states that objects have a value (in addition to type and identity). The value of an object is a rather abstract notion in Python: For example, there is no canonical access method for an object's value. Also, there is no requirement that the value of an object should be constructed in a particular way, e.g. comprised of all its data attributes. Comparison operators implement a particular notion of what the value of an object is. One can think of them as defining the value of an object indirectly, by means of their comparison implementation.

Because all types are (direct or indirect) subtypes of `object`, they inherit the default comparison behavior from `object`. Types can customize their comparison behavior by implementing *rich comparison methods* like `__lt__()`, described in *Customização básica*.

The default behavior for equality comparison (== and `!=`) is based on the identity of the objects. Hence, equality comparison of instances with the same identity results in equality, and equality comparison of instances with different identities results in inequality. A motivation for this default behavior is the desire that all objects should be reflexive (i.e. `x is y` implies `x == y`).

A default order comparison (<, >, <=, and >=) is not provided; an attempt raises `TypeError`. A motivation for this default behavior is the lack of a similar invariant as for equality.

The behavior of the default equality comparison, that instances with different identities are always unequal, may be in contrast to what types will need that have a sensible definition of object value and value-based equality. Such types will need to customize their comparison behavior, and in fact, a number of built-in types have done that.

The following list describes the comparison behavior of the most important built-in types.

- Numbers of built-in numeric types (typesnumeric) and of the standard library types `fractions.Fraction` and `decimal.Decimal` can be compared within and across their types, with the restriction that complex numbers do not support order comparison. Within the limits of the types involved, they compare mathematically (algorithmically) correct without loss of precision.

The not-a-number values `float('NaN')` and `Decimal('NaN')` are special. They are identical to themselves (`x is x` is true) but are not equal to themselves (`x == x` is false). Additionally, comparing any number to a not-a-number value will return `False`. For example, both `3 < float('NaN')` and `float('NaN') < 3` will return `False`.

- Binary sequences (instances of `bytes` or `bytearray`) can be compared within and across their types. They compare lexicographically using the numeric values of their elements.
- Strings (instances of `str`) compare lexicographically using the numerical Unicode code points (the result of the built-in function `ord()`) of their characters.[3]

  Strings and binary sequences cannot be directly compared.
- Sequences (instances of `tuple`, `list`, or `range`) can be compared only within each of their types, with the restriction that ranges do not support order comparison. Equality comparison across these types results in inequality, and ordering comparison across these types raises `TypeError`.

  Sequences compare lexicographically using comparison of corresponding elements, whereby reflexivity of the elements is enforced.

  In enforcing reflexivity of elements, the comparison of collections assumes that for a collection element `x`, `x == x` is always true. Based on that assumption, element identity is compared first, and element comparison is performed only for distinct elements. This approach yields the same result as a strict element comparison would, if the compared elements are reflexive. For non-reflexive elements, the result is different than for strict element comparison, and may be surprising: The non-reflexive not-a-number values for example result in the following comparison behavior when used in a list:

```
>>> nan = float('NaN')
>>> nan is nan
True
>>> nan == nan
False                  <-- the defined non-reflexive behavior of NaN
>>> [nan] == [nan]
True                   <-- list enforces reflexivity and tests identity first
```

  Lexicographical comparison between built-in collections works as follows:

  – For two collections to compare equal, they must be of the same type, have the same length, and each pair of corresponding elements must compare equal (for example, `[1,2] == (1,2)` is false because the type is not the same).

  – Collections that support order comparison are ordered the same as their first unequal elements (for example, `[1,2,x] <= [1,2,y]` has the same value as `x <= y`). If a corresponding element does not exist, the shorter collection is ordered first (for example, `[1,2] < [1,2,3]` is true).
- Mappings (instances of `dict`) compare equal if and only if they have equal *(key, value)* pairs. Equality comparison of the keys and values enforces reflexivity.

  Order comparisons (<, >, <=, and >=) raise `TypeError`.
- Sets (instances of `set` or `frozenset`) can be compared within and across their types.

[3] The Unicode standard distinguishes between *code points* (e.g. U+0041) and *abstract characters* (e.g. "LATIN CAPITAL LETTER A"). While most abstract characters in Unicode are only represented using one code point, there is a number of abstract characters that can in addition be represented using a sequence of more than one code point. For example, the abstract character "LATIN CAPITAL LETTER C WITH CEDILLA" can be represented as a single *precomposed character* at code position U+00C7, or as a sequence of a *base character* at code position U+0043 (LATIN CAPITAL LETTER C), followed by a *combining character* at code position U+0327 (COMBINING CEDILLA).

The comparison operators on strings compare at the level of Unicode code points. This may be counter-intuitive to humans. For example, `"\u00C7" == "\u0043\u0327"` is `False`, even though both strings represent the same abstract character "LATIN CAPITAL LETTER C WITH CEDILLA".

To compare strings at the level of abstract characters (that is, in a way intuitive to humans), use `unicodedata.normalize()`.

They define order comparison operators to mean subset and superset tests. Those relations do not define total orderings (for example, the two sets `{1,2}` and `{2,3}` are not equal, nor subsets of one another, nor supersets of one another). Accordingly, sets are not appropriate arguments for functions which depend on total ordering (for example, `min()`, `max()`, and `sorted()` produce undefined results given a list of sets as inputs).

Comparison of sets enforces reflexivity of its elements.

- Most other built-in types have no comparison methods implemented, so they inherit the default comparison behavior.

User-defined classes that customize their comparison behavior should follow some consistency rules, if possible:

- Equality comparison should be reflexive. In other words, identical objects should compare equal:

  `x is y` implies `x == y`

- Comparison should be symmetric. In other words, the following expressions should have the same result:

  `x == y` and `y == x`

  `x != y` and `y != x`

  `x < y` and `y > x`

  `x <= y` and `y >= x`

- Comparison should be transitive. The following (non-exhaustive) examples illustrate that:

  `x > y and y > z` implies `x > z`

  `x < y and y <= z` implies `x < z`

- Inverse comparison should result in the boolean negation. In other words, the following expressions should have the same result:

  `x == y` and `not x != y`

  `x < y` and `not x >= y` (for total ordering)

  `x > y` and `not x <= y` (for total ordering)

  The last two expressions apply to totally ordered collections (e.g. to sequences, but not to sets or mappings). See also the `total_ordering()` decorator.

- The `hash()` result should be consistent with equality. Objects that are equal should either have the same hash value, or be marked as unhashable.

Python does not enforce these consistency rules. In fact, the not-a-number values are an example for not following these rules.

6.10.2 Membership test operations

The operators *in* and *not in* test for membership. `x in s` evaluates to `True` if *x* is a member of *s*, and `False` otherwise. `x not in s` returns the negation of `x in s`. All built-in sequences and set types support this as well as dictionary, for which *in* tests whether the dictionary has a given key. For container types such as list, tuple, set, frozenset, dict, or collections.deque, the expression `x in y` is equivalent to `any(x is e or x == e for e in y)`.

For the string and bytes types, `x in y` is `True` if and only if *x* is a substring of *y*. An equivalent test is `y.find(x) != -1`. Empty strings are always considered to be a substring of any other string, so `"" in "abc"` will return `True`.

For user-defined classes which define the *__contains__()* method, `x in y` returns `True` if `y.__contains__(x)` returns a true value, and `False` otherwise.

For user-defined classes which do not define *__contains__()* but do define *__iter__()*, `x in y` is `True` if some value `z` with `x == z` is produced while iterating over `y`. If an exception is raised during the iteration, it is as if *in* raised that exception.

Lastly, the old-style iteration protocol is tried: if a class defines *__getitem__()*, `x in y` is `True` if and only if there is a non-negative integer index *i* such that `x == y[i]`, and all lower integer indices do not raise `IndexError` exception. (If any other exception is raised, it is as if *in* raised that exception).

The operator *not in* is defined to have the inverse true value of *in*.

6.10.3 Identity comparisons

The operators *is* and *is not* test for object identity: `x is y` is true if and only if *x* and *y* are the same object. Object identity is determined using the `id()` function. `x is not y` yields the inverse truth value.[4]

6.11 Boolean operations

```
or_test   ::=  and_test | or_test "or" and_test
and_test  ::=  not_test | and_test "and" not_test
not_test  ::=  comparison | "not" not_test
```

In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: `False`, `None`, numeric zero of all types, and empty strings and containers (including strings, tuples, lists, dictionaries, sets and frozensets). All other values are interpreted as true. User-defined objects can customize their truth value by providing a *__bool__()* method.

The operator *not* yields `True` if its argument is false, `False` otherwise.

The expression `x and y` first evaluates *x*; if *x* is false, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

The expression `x or y` first evaluates *x*; if *x* is true, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

Note that neither *and* nor *or* restrict the value and type they return to `False` and `True`, but rather return the last evaluated argument. This is sometimes useful, e.g., if `s` is a string that should be replaced by a default value if it is empty, the expression `s or 'foo'` yields the desired value. Because *not* has to create a new value, it returns a boolean value regardless of the type of its argument (for example, `not 'foo'` produces `False` rather than `''`.)

6.12 Conditional expressions

```
conditional_expression  ::=  or_test ["if" or_test "else" expression]
expression              ::=  conditional_expression | lambda_expr
expression_nocond       ::=  or_test | lambda_expr_nocond
```

Conditional expressions (sometimes called a “ternary operator”) have the lowest priority of all Python operations.

The expression `x if C else y` first evaluates the condition, *C* rather than *x*. If *C* is true, *x* is evaluated and its value is returned; otherwise, *y* is evaluated and its value is returned.

[4] Due to automatic garbage-collection, free lists, and the dynamic nature of descriptors, you may notice seemingly unusual behaviour in certain uses of the *is* operator, like those involving comparisons between instance methods, or constants. Check their documentation for more info.

See **PEP 308** for more details about conditional expressions.

6.13 Lambdas

```
lambda_expr          ::=  "lambda" [parameter_list] ":" expression
lambda_expr_nocond   ::=  "lambda" [parameter_list] ":" expression_nocond
```

Lambda expressions (sometimes called lambda forms) are used to create anonymous functions. The expression `lambda parameters: expression` yields a function object. The unnamed object behaves like a function object defined with:

```
def <lambda>(parameters):
    return expression
```

See section *Definições de função* for the syntax of parameter lists. Note that functions created with lambda expressions cannot contain statements or annotations.

6.14 Expression lists

```
expression_list     ::=  expression ("," expression)* [","]
starred_list        ::=  starred_item ("," starred_item)* [","]
starred_expression  ::=  expression | (starred_item ",")* [starred_item]
starred_item        ::=  expression | "*" or_expr
```

Except when part of a list or set display, an expression list containing at least one comma yields a tuple. The length of the tuple is the number of expressions in the list. The expressions are evaluated from left to right.

An asterisk `*` denotes *iterable unpacking*. Its operand must be an *iterable*. The iterable is expanded into a sequence of items, which are included in the new tuple, list, or set, at the site of the unpacking.

Novo na versão 3.5: Iterable unpacking in expression lists, originally proposed by **PEP 448**.

The trailing comma is required only to create a single tuple (a.k.a. a *singleton*); it is optional in all other cases. A single expression without a trailing comma doesn't create a tuple, but rather yields the value of that expression. (To create an empty tuple, use an empty pair of parentheses: `()`.)

6.15 Evaluation order

Python evaluates expressions from left to right. Notice that while evaluating an assignment, the right-hand side is evaluated before the left-hand side.

In the following lines, expressions will be evaluated in the arithmetic order of their suffixes:

```
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
expr1(expr2, expr3, *expr4, **expr5)
expr3, expr4 = expr1, expr2
```

6.16 Operator precedence

The following table summarizes the operator precedence in Python, from lowest precedence (least binding) to highest precedence (most binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for exponentiation, which groups from right to left).

Note that comparisons, membership tests, and identity tests, all have the same precedence and have a left-to-right chaining feature as described in the *Comparações* section.

Operator	Description (descrição)
`lambda`	Lambda expression
`if` – `else`	Conditional expression
`or`	Boolean OR
`and`	Boolean AND
`not` x	Boolean NOT
`in`, `not in`, `is`, `is not`, <, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, –	Addition and subtraction
*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, remainder[5]
+x, –x, ~x	Positive, negative, bitwise NOT
**	Exponentiation[6]
`await` x	Await expression
x[index], x[index:index], x(arguments...), x.attribute	Subscription, slicing, call, attribute reference
(expressions...), [expressions...], {key: value...}, {expressions...}	Binding or tuple display, list display, dictionary display, set display

Notas de Rodapé

[5] The % operator is also used for string formatting; the same precedence applies.

[6] The power operator ** binds less tightly than an arithmetic or bitwise unary operator on its right, that is, 2**-1 is 0.5.

CAPÍTULO 7

Simple statements

A simple statement is comprised within a single logical line. Several simple statements may occur on a single line separated by semicolons. The syntax for simple statements is:

```
simple_stmt  ::=  expression_stmt
                | assert_stmt
                | assignment_stmt
                | augmented_assignment_stmt
                | annotated_assignment_stmt
                | pass_stmt
                | del_stmt
                | return_stmt
                | yield_stmt
                | raise_stmt
                | break_stmt
                | continue_stmt
                | import_stmt
                | future_stmt
                | global_stmt
                | nonlocal_stmt
```

7.1 Expression statements

Expression statements are used (mostly interactively) to compute and write a value, or (usually) to call a procedure (a function that returns no meaningful result; in Python, procedures return the value `None`). Other uses of expression statements are allowed and occasionally useful. The syntax for an expression statement is:

```
expression_stmt  ::=  starred_expression
```

An expression statement evaluates the expression list (which may be a single expression).

In interactive mode, if the value is not `None`, it is converted to a string using the built-in `repr()` function and the resulting string is written to standard output on a line by itself (except if the result is `None`, so that procedure calls do not cause any output.)

7.2 Assignment statements

Assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects:

```
assignment_stmt  ::=  (target_list "=")+ (starred_expression | yield_expression)
target_list      ::=  target ("," target)* [","]
target           ::=  identifier
                      | "(" [target_list] ")"
                      | "[" [target_list] "]"
                      | attributeref
                      | subscription
                      | slicing
                      | "*" target
```

(See section *Primaries* for the syntax definitions for *attributeref*, *subscription*, and *slicing*.)

An assignment statement evaluates the expression list (remember that this can be a single expression or a comma-separated list, the latter yielding a tuple) and assigns the single resulting object to each of the target lists, from left to right.

Assignment is defined recursively depending on the form of the target (list). When a target is part of a mutable object (an attribute reference, subscription or slicing), the mutable object must ultimately perform the assignment and decide about its validity, and may raise an exception if the assignment is unacceptable. The rules observed by various types and the exceptions raised are given with the definition of the object types (see section *A hierarquia de tipos padrão*).

Assignment of an object to a target list, optionally enclosed in parentheses or square brackets, is recursively defined as follows.

- If the target list is a single target with no trailing comma, optionally in parentheses, the object is assigned to that target.
- Else: The object must be an iterable with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets.
  - If the target list contains one target prefixed with an asterisk, called a “starred” target: The object must be an iterable with at least as many items as there are targets in the target list, minus one. The first items of the iterable are assigned, from left to right, to the targets before the starred target. The final items of the iterable are assigned to the targets after the starred target. A list of the remaining items in the iterable is then assigned to the starred target (the list can be empty).
  - Else: The object must be an iterable with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets.

Assignment of an object to a single target is recursively defined as follows.

- If the target is an identifier (name):
  - If the name does not occur in a *global* or *nonlocal* statement in the current code block: the name is bound to the object in the current local namespace.
  - Otherwise: the name is bound to the object in the global namespace or the outer namespace determined by *nonlocal*, respectively.

The name is rebound if it was already bound. This may cause the reference count for the object previously bound to the name to reach zero, causing the object to be deallocated and its destructor (if it has one) to be called.

- If the target is an attribute reference: The primary expression in the reference is evaluated. It should yield an object with assignable attributes; if this is not the case, `TypeError` is raised. That object is then asked to assign the assigned object to the given attribute; if it cannot perform the assignment, it raises an exception (usually but not necessarily `AttributeError`).

  Note: If the object is a class instance and the attribute reference occurs on both sides of the assignment operator, the RHS expression, `a.x` can access either an instance attribute or (if no instance attribute exists) a class attribute. The LHS target `a.x` is always set as an instance attribute, creating it if necessary. Thus, the two occurrences of `a.x` do not necessarily refer to the same attribute: if the RHS expression refers to a class attribute, the LHS creates a new instance attribute as the target of the assignment:

```
class Cls:
    x = 3             # class variable
inst = Cls()
inst.x = inst.x + 1   # writes inst.x as 4 leaving Cls.x as 3
```

  This description does not necessarily apply to descriptor attributes, such as properties created with `property()`.

- If the target is a subscription: The primary expression in the reference is evaluated. It should yield either a mutable sequence object (such as a list) or a mapping object (such as a dictionary). Next, the subscript expression is evaluated.

  If the primary is a mutable sequence object (such as a list), the subscript must yield an integer. If it is negative, the sequence's length is added to it. The resulting value must be a nonnegative integer less than the sequence's length, and the sequence is asked to assign the assigned object to its item with that index. If the index is out of range, `IndexError` is raised (assignment to a subscripted sequence cannot add new items to a list).

  If the primary is a mapping object (such as a dictionary), the subscript must have a type compatible with the mapping's key type, and the mapping is then asked to create a key/datum pair which maps the subscript to the assigned object. This can either replace an existing key/value pair with the same key value, or insert a new key/value pair (if no key with the same value existed).

  For user-defined objects, the *__setitem__()* method is called with appropriate arguments.

- If the target is a slicing: The primary expression in the reference is evaluated. It should yield a mutable sequence object (such as a list). The assigned object should be a sequence object of the same type. Next, the lower and upper bound expressions are evaluated, insofar they are present; defaults are zero and the sequence's length. The bounds should evaluate to integers. If either bound is negative, the sequence's length is added to it. The resulting bounds are clipped to lie between zero and the sequence's length, inclusive. Finally, the sequence object is asked to replace the slice with the items of the assigned sequence. The length of the slice may be different from the length of the assigned sequence, thus changing the length of the target sequence, if the target sequence allows it.

CPython implementation detail: In the current implementation, the syntax for targets is taken to be the same as for expressions, and invalid syntax is rejected during the code generation phase, causing less detailed error messages.

Although the definition of assignment implies that overlaps between the left-hand side and the right-hand side are 'simultaneous' (for example `a, b = b, a` swaps two variables), overlaps *within* the collection of assigned-to variables occur left-to-right, sometimes resulting in confusion. For instance, the following program prints `[0, 2]`:

```
x = [0, 1]
i = 0
i, x[i] = 1, 2          # i is updated, then x[i] is updated
print(x)
```

Ver também:

PEP 3132 - Extended Iterable Unpacking The specification for the `*target` feature.

7.2.1 Augmented assignment statements

Augmented assignment is the combination, in a single statement, of a binary operation and an assignment statement:

```
augmented_assignment_stmt   ::=   augtarget augop (expression_list | yield_expression)
augtarget                   ::=   identifier | attributeref | subscription | slicing
augop                       ::=   "+=" | "-=" | "*=" | "@=" | "/=" | "//=" | "%=" | "**="
                                  | ">>=" | "<<=" | "&=" | "^=" | "|="
```

(See section *Primaries* for the syntax definitions of the last three symbols.)

An augmented assignment evaluates the target (which, unlike normal assignment statements, cannot be an unpacking) and the expression list, performs the binary operation specific to the type of assignment on the two operands, and assigns the result to the original target. The target is only evaluated once.

An augmented assignment expression like `x += 1` can be rewritten as `x = x + 1` to achieve a similar, but not exactly equal effect. In the augmented version, `x` is only evaluated once. Also, when possible, the actual operation is performed *in-place*, meaning that rather than creating a new object and assigning that to the target, the old object is modified instead.

Unlike normal assignments, augmented assignments evaluate the left-hand side *before* evaluating the right-hand side. For example, `a[i] += f(x)` first looks-up `a[i]`, then it evaluates `f(x)` and performs the addition, and lastly, it writes the result back to `a[i]`.

With the exception of assigning to tuples and multiple targets in a single statement, the assignment done by augmented assignment statements is handled the same way as normal assignments. Similarly, with the exception of the possible *in-place* behavior, the binary operation performed by augmented assignment is the same as the normal binary operations.

For targets which are attribute references, the same *caveat about class and instance attributes* applies as for regular assignments.

7.2.2 Annotated assignment statements

Annotation assignment is the combination, in a single statement, of a variable or attribute annotation and an optional assignment statement:

```
annotated_assignment_stmt   ::=   augtarget ":" expression ["=" expression]
```

The difference from normal *Assignment statements* is that only single target and only single right hand side value is allowed.

For simple names as assignment targets, if in class or module scope, the annotations are evaluated and stored in a special class or module attribute `__annotations__` that is a dictionary mapping from variable names (mangled if private) to evaluated annotations. This attribute is writable and is automatically created at the start of class or module body execution, if annotations are found statically.

For expressions as assignment targets, the annotations are evaluated if in class or module scope, but not stored.

If a name is annotated in a function scope, then this name is local for that scope. Annotations are never evaluated and stored in function scopes.

If the right hand side is present, an annotated assignment performs the actual assignment before evaluating annotations (where applicable). If the right hand side is not present for an expression target, then the interpreter evaluates the target except for the last *__setitem__()* or *__setattr__()* call.

Ver também:

PEP 526 - Syntax for Variable Annotations The proposal that added syntax for annotating the types of variables (in-

cluding class variables and instance variables), instead of expressing them through comments.

PEP 484 - Type hints The proposal that added the `typing` module to provide a standard syntax for type annotations that can be used in static analysis tools and IDEs.

7.3 The `assert` statement

Assert statements are a convenient way to insert debugging assertions into a program:

```
assert_stmt  ::=  "assert" expression ["," expression]
```

The simple form, `assert expression`, is equivalent to

```
if __debug__:
    if not expression: raise AssertionError
```

The extended form, `assert expression1, expression2`, is equivalent to

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

These equivalences assume that `__debug__` and `AssertionError` refer to the built-in variables with those names. In the current implementation, the built-in variable `__debug__` is `True` under normal circumstances, `False` when optimization is requested (command line option `-O`). The current code generator emits no code for an assert statement when optimization is requested at compile time. Note that it is unnecessary to include the source code for the expression that failed in the error message; it will be displayed as part of the stack trace.

Assignments to `__debug__` are illegal. The value for the built-in variable is determined when the interpreter starts.

7.4 The `pass` statement

```
pass_stmt  ::=  "pass"
```

pass is a null operation — when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed, for example:

```
def f(arg): pass    # a function that does nothing (yet)

class C: pass       # a class with no methods (yet)
```

7.5 O comando `del`

```
del_stmt   ::=   "del" target_list
```

Deletion is recursively defined very similar to the way assignment is defined. Rather than spelling it out in full details, here are some hints.

Deletion of a target list recursively deletes each target, from left to right.

Deletion of a name removes the binding of that name from the local or global namespace, depending on whether the name occurs in a *global* statement in the same code block. If the name is unbound, a `NameError` exception will be raised.

Deletion of attribute references, subscriptions and slicings is passed to the primary object involved; deletion of a slicing is in general equivalent to assignment of an empty slice of the right type (but even this is determined by the sliced object).

Alterado na versão 3.2: Previously it was illegal to delete a name from the local namespace if it occurs as a free variable in a nested block.

7.6 The `return` statement

```
return_stmt   ::=   "return" [expression_list]
```

return may only occur syntactically nested in a function definition, not within a nested class definition.

If an expression list is present, it is evaluated, else `None` is substituted.

return leaves the current function call with the expression list (or `None`) as return value.

When *return* passes control out of a *try* statement with a *finally* clause, that *finally* clause is executed before really leaving the function.

In a generator function, the *return* statement indicates that the generator is done and will cause `StopIteration` to be raised. The returned value (if any) is used as an argument to construct `StopIteration` and becomes the `StopIteration.value` attribute.

In an asynchronous generator function, an empty *return* statement indicates that the asynchronous generator is done and will cause `StopAsyncIteration` to be raised. A non-empty *return* statement is a syntax error in an asynchronous generator function.

7.7 A declaração `yield`

```
yield_stmt   ::=   yield_expression
```

A *yield* statement is semantically equivalent to a *yield expression*. The yield statement can be used to omit the parentheses that would otherwise be required in the equivalent yield expression statement. For example, the yield statements

```
yield <expr>
yield from <expr>
```

are equivalent to the yield expression statements

```
(yield <expr>)
(yield from <expr>)
```

Yield expressions and statements are only used when defining a *generator* function, and are only used in the body of the generator function. Using yield in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

For full details of `yield` semantics, refer to the *Yield expressions* section.

7.8 The `raise` statement

```
raise_stmt  ::=  "raise" [expression ["from" expression]]
```

If no expressions are present, `raise` re-raises the last exception that was active in the current scope. If no exception is active in the current scope, a `RuntimeError` exception is raised indicating that this is an error.

Otherwise, `raise` evaluates the first expression as the exception object. It must be either a subclass or an instance of `BaseException`. If it is a class, the exception instance will be obtained when needed by instantiating the class with no arguments.

The *type* of the exception is the exception instance's class, the *value* is the instance itself.

A traceback object is normally created automatically when an exception is raised and attached to it as the `__traceback__` attribute, which is writable. You can create an exception and set your own traceback in one step using the `with_traceback()` exception method (which returns the same exception instance, with its traceback set to its argument), like so:

```
raise Exception("foo occurred").with_traceback(tracebackobj)
```

The `from` clause is used for exception chaining: if given, the second *expression* must be another exception class or instance, which will then be attached to the raised exception as the `__cause__` attribute (which is writable). If the raised exception is not handled, both exceptions will be printed:

```
>>> try:
...     print(1 / 0)
... except Exception as exc:
...     raise RuntimeError("Something bad happened") from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

A similar mechanism works implicitly if an exception is raised inside an exception handler or a `finally` clause: the previous exception is then attached as the new exception's `__context__` attribute:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

(continua na próxima página)

(continuação da página anterior)

```
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

Exception chaining can be explicitly suppressed by specifying `None` in the `from` clause:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened") from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

Additional information on exceptions can be found in section *Exceções*, and information about handling exceptions is in section *The try statement*.

Alterado na versão 3.3: `None` is now permitted as `Y` in `raise X from Y`.

Novo na versão 3.3: The `__suppress_context__` attribute to suppress automatic display of the exception context.

7.9 The `break` statement

```
break_stmt  ::=  "break"
```

break may only occur syntactically nested in a *for* or *while* loop, but not nested in a function or class definition within that loop.

It terminates the nearest enclosing loop, skipping the optional *else* clause if the loop has one.

If a *for* loop is terminated by *break*, the loop control target keeps its current value.

When *break* passes control out of a *try* statement with a *finally* clause, that *finally* clause is executed before really leaving the loop.

7.10 The `continue` statement

```
continue_stmt  ::=  "continue"
```

continue may only occur syntactically nested in a *for* or *while* loop, but not nested in a function or class definition or *finally* clause within that loop. It continues with the next cycle of the nearest enclosing loop.

When *continue* passes control out of a *try* statement with a *finally* clause, that *finally* clause is executed before really starting the next loop cycle.

7.11 The `import` statement

```
import_stmt     ::=  "import" module ["as" identifier] ("," module ["as" identifier])*
                     | "from" relative_module "import" identifier ["as" identifier]
                     ("," identifier ["as" identifier])*
                     | "from" relative_module "import" "(" identifier ["as" identifier]
                     ("," identifier ["as" identifier])* [","] ")"
                     | "from" module "import" "*"
module          ::=  (identifier ".")* identifier
relative_module ::=  "."* module | "."+
```

The basic import statement (no *from* clause) is executed in two steps:

1. find a module, loading and initializing it if necessary
2. define a name or names in the local namespace for the scope where the *import* statement occurs.

When the statement contains multiple clauses (separated by commas) the two steps are carried out separately for each clause, just as though the clauses had been separated out into individual import statements.

The details of the first step, finding and loading modules are described in greater detail in the section on the *import system*, which also describes the various types of packages and modules that can be imported, as well as all the hooks that can be used to customize the import system. Note that failures in this step may indicate either that the module could not be located, *or* that an error occurred while initializing the module, which includes execution of the module's code.

If the requested module is retrieved successfully, it will be made available in the local namespace in one of three ways:

- If the module name is followed by *as*, then the name following *as* is bound directly to the imported module.
- If no other name is specified, and the module being imported is a top level module, the module's name is bound in the local namespace as a reference to the imported module
- If the module being imported is *not* a top level module, then the name of the top level package that contains the module is bound in the local namespace as a reference to the top level package. The imported module must be accessed using its full qualified name rather than directly

The *from* form uses a slightly more complex process:

1. find the module specified in the *from* clause, loading and initializing it if necessary;
2. for each of the identifiers specified in the *import* clauses:
    1. check if the imported module has an attribute by that name
    2. if not, attempt to import a submodule with that name and then check the imported module again for that attribute
    3. if the attribute is not found, `ImportError` is raised.
    4. otherwise, a reference to that value is stored in the local namespace, using the name in the *as* clause if it is present, otherwise using the attribute name

Exemplos:

```
import foo                 # foo imported and bound locally
import foo.bar.baz         # foo.bar.baz imported, foo bound locally
import foo.bar.baz as fbb  # foo.bar.baz imported and bound as fbb
from foo.bar import baz    # foo.bar.baz imported and bound as baz
from foo import attr       # foo imported and foo.attr bound as attr
```

If the list of identifiers is replaced by a star (`'*'`), all public names defined in the module are bound in the local namespace for the scope where the *import* statement occurs.

The *public names* defined by a module are determined by checking the module's namespace for a variable named `__all__`; if defined, it must be a sequence of strings which are names defined or imported by that module. The names given in `__all__` are all considered public and are required to exist. If `__all__` is not defined, the set of public names includes all names found in the module's namespace which do not begin with an underscore character (`'_'`). `__all__` should contain the entire public API. It is intended to avoid accidentally exporting items that are not part of the API (such as library modules which were imported and used within the module).

The wild card form of import — `from module import *` — is only allowed at the module level. Attempting to use it in class or function definitions will raise a `SyntaxError`.

When specifying what module to import you do not have to specify the absolute name of the module. When a module or package is contained within another package it is possible to make a relative import within the same top package without having to mention the package name. By using leading dots in the specified module or package after *from* you can specify how high to traverse up the current package hierarchy without specifying exact names. One leading dot means the current package where the module making the import exists. Two dots means up one package level. Three dots is up two levels, etc. So if you execute `from . import mod` from a module in the `pkg` package then you will end up importing `pkg.mod`. If you execute `from ..subpkg2 import mod` from within `pkg.subpkg1` you will import `pkg.subpkg2.mod`. The specification for relative imports is contained within **PEP 328**.

`importlib.import_module()` is provided to support applications that determine dynamically the modules to be loaded.

7.11.1 Future statements

A *future statement* is a directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python where the feature becomes standard.

The future statement is intended to ease migration to future versions of Python that introduce incompatible changes to the language. It allows use of the new features on a per-module basis before the release in which the feature becomes standard.

```
future_stmt   ::=   "from" "__future__" "import" feature ["as" identifier]
                    ("," feature ["as" identifier])*
                    | "from" "__future__" "import" "(" feature ["as" identifier]
                    ("," feature ["as" identifier])* [","] ")"
feature       ::=   identifier
```

A future statement must appear near the top of the module. The only lines that can appear before a future statement are:

- the module docstring (if any),
- comments,
- blank lines, and
- other future statements.

The features recognized by Python 3.0 are `absolute_import`, `division`, `generators`, `unicode_literals`, `print_function`, `nested_scopes` and `with_statement`. They are all redundant because they are always enabled, and only kept for backwards compatibility.

A future statement is recognized and treated specially at compile time: Changes to the semantics of core constructs are often implemented by generating different code. It may even be the case that a new feature introduces new incompatible syntax (such as a new reserved word), in which case the compiler may need to parse the module differently. Such decisions

cannot be pushed off until runtime.

For any given release, the compiler knows which feature names have been defined, and raises a compile-time error if a future statement contains a feature not known to it.

The direct runtime semantics are the same as for any import statement: there is a standard module `__future__`, described later, and it will be imported in the usual way at the time the future statement is executed.

The interesting runtime semantics depend on the specific feature enabled by the future statement.

Note that there is nothing special about the statement:

```
import __future__ [as name]
```

That is not a future statement; it's an ordinary import statement with no special semantics or syntax restrictions.

Code compiled by calls to the built-in functions `exec()` and `compile()` that occur in a module `M` containing a future statement will, by default, use the new syntax or semantics associated with the future statement. This can be controlled by optional arguments to `compile()` — see the documentation of that function for details.

A future statement typed at an interactive interpreter prompt will take effect for the rest of the interpreter session. If an interpreter is started with the `-i` option, is passed a script name to execute, and the script includes a future statement, it will be in effect in the interactive session started after the script is executed.

Ver também:

PEP 236 - Back to the __future__ The original proposal for the __future__ mechanism.

7.12 The `global` statement

```
global_stmt  ::=  "global" identifier ("," identifier)*
```

The *global* statement is a declaration which holds for the entire current code block. It means that the listed identifiers are to be interpreted as globals. It would be impossible to assign to a global variable without *global*, although free variables may refer to globals without being declared global.

Names listed in a *global* statement must not be used in the same code block textually preceding that *global* statement.

Names listed in a *global* statement must not be defined as formal parameters or in a *for* loop control target, *class* definition, function definition, *import* statement, or variable annotation.

CPython implementation detail: The current implementation does not enforce some of these restrictions, but programs should not abuse this freedom, as future implementations may enforce them or silently change the meaning of the program.

Programmer's note: *global* is a directive to the parser. It applies only to code parsed at the same time as the *global* statement. In particular, a *global* statement contained in a string or code object supplied to the built-in `exec()` function does not affect the code block *containing* the function call, and code contained in such a string is unaffected by *global* statements in the code containing the function call. The same applies to the `eval()` and `compile()` functions.

7.13 The `nonlocal` statement

```
nonlocal_stmt  ::=  "nonlocal" identifier ("," identifier)*
```

The *nonlocal* statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding globals. This is important because the default behavior for binding is to search the local namespace first. The statement allows encapsulated code to rebind variables outside of the local scope besides the global (module) scope.

Names listed in a *nonlocal* statement, unlike those listed in a *global* statement, must refer to pre-existing bindings in an enclosing scope (the scope in which a new binding should be created cannot be determined unambiguously).

Names listed in a *nonlocal* statement must not collide with pre-existing bindings in the local scope.

Ver também:

PEP 3104 - Access to Names in Outer Scopes The specification for the *nonlocal* statement.

CAPÍTULO 8

Declarações compostas

Declarações compostas contém (grupos de) outras declarações; Elas afetam ou controlam a execução dessas outras declarações de alguma maneira. Em geral, declarações compostas abrangem múltiplas linhas, no entanto em algumas manifestações simples uma declaração composta inteira pode estar contida em uma linha.

As instruções *if*, *while* e *for* implementam construções tradicionais de controle do fluxo de execução. *try* especifica tratadores de exceções e/ou código de limpeza para um grupo de instruções, enquanto a palavra reservada *with* permite a execução de código de inicialização e finalização em volta de um bloco de código. Definições de função e classe também são sintaticamente instruções compostas.

A compound statement consists of one or more 'clauses.' A clause consists of a header and a 'suite.' The clause headers of a particular compound statement are all at the same indentation level. Each clause header begins with a uniquely identifying keyword and ends with a colon. A suite is a group of statements controlled by a clause. A suite can be one or more semicolon-separated simple statements on the same line as the header, following the header's colon, or it can be one or more indented statements on subsequent lines. Only the latter form of a suite can contain nested compound statements; the following is illegal, mostly because it wouldn't be clear to which *if* clause a following *else* clause would belong:

```
if test1: if test2: print(x)
```

Also note that the semicolon binds tighter than the colon in this context, so that in the following example, either all or none of the `print()` calls are executed:

```
if x < y < z: print(x); print(y); print(z)
```

Summarizing:

```
compound_stmt   ::=   if_stmt
                    | while_stmt
                    | for_stmt
                    | try_stmt
                    | with_stmt
                    | funcdef
                    | classdef
                    | async_with_stmt
```

```
                    | async_for_stmt
                    | async_funcdef
suite           ::=  stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement       ::=  stmt_list NEWLINE | compound_stmt
stmt_list       ::=  simple_stmt (";" simple_stmt)* [";"]
```

Note que instruções sempre terminam em uma `NEWLINE` possivelmente seguida por uma `DEDENT`. Note também que cláusulas de continuação sempre começam com uma palavra reservada que não pode iniciar uma instrução, desta forma não há ambiguidades (o problema do *else*' pendurado é resolvido em Python obrigando que instruções *if* aninhadas tenham indentação)

A formatação das regras de gramática nas próximas seções põe cada cláusula em uma linha separada para as tornar mais claras.

8.1 The `if` statement

The *if* statement is used for conditional execution:

```
if_stmt  ::=  "if" expression ":" suite
              ("elif" expression ":" suite)*
              ["else" ":" suite]
```

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true (see section *Boolean operations* for the definition of true and false); then that suite is executed (and no other part of the *if* statement is executed or evaluated). If all expressions are false, the suite of the *else* clause, if present, is executed.

8.2 The `while` statement

The *while* statement is used for repeated execution as long as an expression is true:

```
while_stmt  ::=  "while" expression ":" suite
                 ["else" ":" suite]
```

This repeatedly tests the expression and, if it is true, executes the first suite; if the expression is false (which may be the first time it is tested) the suite of the *else* clause, if present, is executed and the loop terminates.

A *break* statement executed in the first suite terminates the loop without executing the *else* clause's suite. A *continue* statement executed in the first suite skips the rest of the suite and goes back to testing the expression.

8.3 The `for` statement

The *for* statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:

```
for_stmt  ::=  "for" target_list "in" expression_list ":" suite
               ["else" ":" suite]
```

The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the `expression_list`. The suite is then executed once for each item provided by the iterator, in the order returned by the iterator. Each item in turn is assigned to the target list using the standard rules for assignments (see *Assignment statements*), and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty or an iterator raises a `StopIteration` exception), the suite in the *else* clause, if present, is executed, and the loop terminates.

A *break* statement executed in the first suite terminates the loop without executing the *else* clause's suite. A *continue* statement executed in the first suite skips the rest of the suite and continues with the next item, or with the *else* clause if there is no next item.

The for-loop makes assignments to the variables(s) in the target list. This overwrites all previous assignments to those variables including those made in the suite of the for-loop:

```
for i in range(10):
    print(i)
    i = 5             # this will not affect the for-loop
                      # because i will be overwritten with the next
                      # index in the range
```

Names in the target list are not deleted when the loop is finished, but if the sequence is empty, they will not have been assigned to at all by the loop. Hint: the built-in function `range()` returns an iterator of integers suitable to emulate the effect of Pascal's `for i := a to b do`; e.g., `list(range(3))` returns the list `[0, 1, 2]`.

Nota: There is a subtlety when the sequence is being modified by the loop (this can only occur for mutable sequences, e.g. lists). An internal counter is used to keep track of which item is used next, and this is incremented on each iteration. When this counter has reached the length of the sequence the loop terminates. This means that if the suite deletes the current (or a previous) item from the sequence, the next item will be skipped (since it gets the index of the current item which has already been treated). Likewise, if the suite inserts an item in the sequence before the current item, the current item will be treated again the next time through the loop. This can lead to nasty bugs that can be avoided by making a temporary copy using a slice of the whole sequence, e.g.,

```
for x in a[:]:
    if x < 0: a.remove(x)
```

8.4 The `try` statement

The *try* statement specifies exception handlers and/or cleanup code for a group of statements:

```
try_stmt   ::=  try1_stmt | try2_stmt
try1_stmt  ::=  "try" ":" suite
                ("except" [expression ["as" identifier]] ":" suite)+
                ["else" ":" suite]
                ["finally" ":" suite]
try2_stmt  ::=  "try" ":" suite
                "finally" ":" suite
```

The *except* clause(s) specify one or more exception handlers. When no exception occurs in the *try* clause, no exception handler is executed. When an exception occurs in the *try* suite, a search for an exception handler is started. This search inspects the except clauses in turn until one is found that matches the exception. An expression-less except clause, if present, must be last; it matches any exception. For an except clause with an expression, that expression is evaluated, and the clause matches the exception if the resulting object is "compatible" with the exception. An object is compatible with an exception if it is the class or a base class of the exception object or a tuple containing an item compatible with the exception.

If no except clause matches the exception, the search for an exception handler continues in the surrounding code and on the invocation stack.[1]

If the evaluation of an expression in the header of an except clause raises an exception, the original search for a handler is canceled and a search starts for the new exception in the surrounding code and on the call stack (it is treated as if the entire *try* statement raised the exception).

When a matching except clause is found, the exception is assigned to the target specified after the *as* keyword in that except clause, if present, and the except clause's suite is executed. All except clauses must have an executable block. When the end of this block is reached, execution continues normally after the entire try statement. (This means that if two nested handlers exist for the same exception, and the exception occurs in the try clause of the inner handler, the outer handler will not handle the exception.)

When an exception has been assigned using `as target`, it is cleared at the end of the except clause. This is as if

```
except E as N:
    foo
```

was translated to

```
except E as N:
    try:
        foo
    finally:
        del N
```

This means the exception must be assigned to a different name to be able to refer to it after the except clause. Exceptions are cleared because with the traceback attached to them, they form a reference cycle with the stack frame, keeping all locals in that frame alive until the next garbage collection occurs.

Before an except clause's suite is executed, details about the exception are stored in the `sys` module and can be accessed via `sys.exc_info()`. `sys.exc_info()` returns a 3-tuple consisting of the exception class, the exception instance

[1] The exception is propagated to the invocation stack unless there is a *finally* clause which happens to raise another exception. That new exception causes the old one to be lost.

and a traceback object (see section *A hierarquia de tipos padrão*) identifying the point in the program where the exception occurred. `sys.exc_info()` values are restored to their previous values (before the call) when returning from a function that handled an exception.

The optional `else` clause is executed if the control flow leaves the `try` suite, no exception was raised, and no `return`, `continue`, or `break` statement was executed. Exceptions in the `else` clause are not handled by the preceding `except` clauses.

If `finally` is present, it specifies a 'cleanup' handler. The `try` clause is executed, including any `except` and `else` clauses. If an exception occurs in any of the clauses and is not handled, the exception is temporarily saved. The `finally` clause is executed. If there is a saved exception it is re-raised at the end of the `finally` clause. If the `finally` clause raises another exception, the saved exception is set as the context of the new exception. If the `finally` clause executes a `return` or `break` statement, the saved exception is discarded:

```
>>> def f():
...     try:
...         1/0
...     finally:
...         return 42
...
>>> f()
42
```

The exception information is not available to the program during execution of the `finally` clause.

When a `return`, `break` or `continue` statement is executed in the `try` suite of a `try`…`finally` statement, the `finally` clause is also executed 'on the way out.' A `continue` statement is illegal in the `finally` clause. (The reason is a problem with the current implementation — this restriction may be lifted in the future).

The return value of a function is determined by the last `return` statement executed. Since the `finally` clause always executes, a `return` statement executed in the `finally` clause will always be the last one executed:

```
>>> def foo():
...     try:
...         return 'try'
...     finally:
...         return 'finally'
...
>>> foo()
'finally'
```

Additional information on exceptions can be found in section *Exceções*, and information on using the `raise` statement to generate exceptions may be found in section *The raise statement*.

8.5 The `with` statement

The `with` statement is used to wrap the execution of a block with methods defined by a context manager (see section *Com gerenciadores de contexto de instruções*). This allows common `try`…`except`…`finally` usage patterns to be encapsulated for convenient reuse.

```
with_stmt   ::=   "with" with_item (","  with_item)* ":" suite
with_item   ::=   expression ["as" target]
```

The execution of the `with` statement with one "item" proceeds as follows:

1. The context expression (the expression given in the *with_item*) is evaluated to obtain a context manager.
2. The context manager's *__exit__()* is loaded for later use.
3. The context manager's *__enter__()* method is invoked.
4. If a target was included in the *with* statement, the return value from *__enter__()* is assigned to it.

   Nota: The *with* statement guarantees that if the *__enter__()* method returns without an error, then *__exit__()* will always be called. Thus, if an error occurs during the assignment to the target list, it will be treated the same as an error occurring within the suite would be. See step 6 below.

5. The suite is executed.
6. The context manager's *__exit__()* method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to *__exit__()*. Otherwise, three `None` arguments are supplied.

   If the suite was exited due to an exception, and the return value from the *__exit__()* method was false, the exception is reraised. If the return value was true, the exception is suppressed, and execution continues with the statement following the *with* statement.

   If the suite was exited for any reason other than an exception, the return value from *__exit__()* is ignored, and execution proceeds at the normal location for the kind of exit that was taken.

With more than one item, the context managers are processed as if multiple *with* statements were nested:

```
with A() as a, B() as b:
    suite
```

é equivalente a:

```
with A() as a:
    with B() as b:
        suite
```

Alterado na versão 3.1: Support for multiple context expressions.

Ver também:

PEP 343 - The "with" statement A especificação, o histórico e os exemplos para a instrução Python *with*.

8.6 Definições de função

A function definition defines a user-defined function object (see section *A hierarquia de tipos padrão*):

```
funcdef                 ::=  [decorators] "def" funcname "(" [parameter_list] ")"
                             ["->" expression] ":" suite
decorators              ::=  decorator+
decorator               ::=  "@" dotted_name ["(" [argument_list [","]] ")"] NEWLINE
dotted_name             ::=  identifier ("." identifier)*
parameter_list          ::=  defparameter ("," defparameter)* ["," [parameter_list_stara
                             | parameter_list_starargs
parameter_list_starargs ::=  "*" [parameter] ("," defparameter)* ["," ["**" parameter [",
                             | "**" parameter [","]
parameter               ::=  identifier [":" expression]
```

```
defparameter              ::=   parameter ["=" expression]
funcname                  ::=   identifier
```

A function definition is an executable statement. Its execution binds the function name in the current local namespace to a function object (a wrapper around the executable code for the function). This function object contains a reference to the current global namespace as the global namespace to be used when the function is called.

The function definition does not execute the function body; this gets executed only when the function is called.[2]

A function definition may be wrapped by one or more *decorator* expressions. Decorator expressions are evaluated when the function is defined, in the scope that contains the function definition. The result must be a callable, which is invoked with the function object as the only argument. The returned value is bound to the function name instead of the function object. Multiple decorators are applied in nested fashion. For example, the following code

```
@f1(arg)
@f2
def func(): pass
```

is roughly equivalent to

```
def func(): pass
func = f1(arg)(f2(func))
```

except that the original function is not temporarily bound to the name `func`.

When one or more *parameters* have the form *parameter* = *expression*, the function is said to have "default parameter values." For a parameter with a default value, the corresponding *argument* may be omitted from a call, in which case the parameter's default value is substituted. If a parameter has a default value, all following parameters up until the "`*`" must also have a default value — this is a syntactic restriction that is not expressed by the grammar.

Default parameter values are evaluated from left to right when the function definition is executed. This means that the expression is evaluated once, when the function is defined, and that the same "pre-computed" value is used for each call. This is especially important to understand when a default parameter is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default value is in effect modified. This is generally not what was intended. A way around this is to use `None` as the default, and explicitly test for it in the body of the function, e.g.:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Function call semantics are described in more detail in section *Calls*. A function call always assigns values to all parameters mentioned in the parameter list, either from position arguments, from keyword arguments, or from default values. If the form "`*identifier`" is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form "`**identifier`" is present, it is initialized to a new ordered mapping receiving any excess keyword arguments, defaulting to a new empty mapping of the same type. Parameters after "`*`" or "`*identifier`" are keyword-only parameters and may only be passed used keyword arguments.

Parameters may have annotations of the form "`: expression`" following the parameter name. Any parameter may have an annotation even those of the form `*identifier` or `**identifier`. Functions may have "return" annotation of the form "`-> expression`" after the parameter list. These annotations can be any valid Python expression and are evaluated when the function definition is executed. Annotations may be evaluated in a different order than they appear in the source code. The presence of annotations does not change the semantics of a function. The annotation values are

[2] A string literal appearing as the first statement in the function body is transformed into the function's `__doc__` attribute and therefore the function's *docstring*.

available as values of a dictionary keyed by the parameters' names in the __annotations__ attribute of the function object.

It is also possible to create anonymous functions (functions not bound to a name), for immediate use in expressions. This uses lambda expressions, described in section *Lambdas*. Note that the lambda expression is merely a shorthand for a simplified function definition; a function defined in a "def" statement can be passed around or assigned to another name just like a function defined by a lambda expression. The "def" form is actually more powerful since it allows the execution of multiple statements and annotations.

Programmer's note: Functions are first-class objects. A "def" statement executed inside a function definition defines a local function that can be returned or passed around. Free variables used in the nested function can access the local variables of the function containing the def. See section *Nomeação e ligação* for details.

Ver também:

PEP 3107 - Function Annotations The original specification for function annotations.

8.7 Definições de classe

A class definition defines a class object (see section *A hierarquia de tipos padrão*):

```
classdef    ::=  [decorators] "class" classname [inheritance] ":" suite
inheritance ::=  "(" [argument_list] ")"
classname   ::=  identifier
```

A class definition is an executable statement. The inheritance list usually gives a list of base classes (see *Metaclasses* for more advanced uses), so each item in the list should evaluate to a class object which allows subclassing. Classes without an inheritance list inherit, by default, from the base class object; hence,

```
class Foo:
    pass
```

é equivalente a:

```
class Foo(object):
    pass
```

The class's suite is then executed in a new execution frame (see *Nomeação e ligação*), using a newly created local namespace and the original global namespace. (Usually, the suite contains mostly function definitions.) When the class's suite finishes execution, its execution frame is discarded but its local namespace is saved.[3] A class object is then created using the inheritance list for the base classes and the saved local namespace for the attribute dictionary. The class name is bound to this class object in the original local namespace.

The order in which attributes are defined in the class body is preserved in the new class's __dict__. Note that this is reliable only right after the class is created and only for classes that were defined using the definition syntax.

Class creation can be customized heavily using *metaclasses*.

Classes can also be decorated: just like when decorating functions,

```
@f1(arg)
@f2
class Foo: pass
```

[3] A string literal appearing as the first statement in the class body is transformed into the namespace's __doc__ item and therefore the class's *docstring*.

is roughly equivalent to

```
class Foo: pass
Foo = f1(arg)(f2(Foo))
```

The evaluation rules for the decorator expressions are the same as for function decorators. The result is then bound to the class name.

Programmer's note: Variables defined in the class definition are class attributes; they are shared by instances. Instance attributes can be set in a method with `self.name = value`. Both class and instance attributes are accessible through the notation "`self.name`", and an instance attribute hides a class attribute with the same name when accessed in this way. Class attributes can be used as defaults for instance attributes, but using mutable values there can lead to unexpected results. *Descriptors* can be used to create instance variables with different implementation details.

Ver também:

PEP 3115 - Metaclasses no Python 3000 The proposal that changed the declaration of metaclasses to the current syntax, and the semantics for how classes with metaclasses are constructed.

PEP 3129 - Class Decorators The proposal that added class decorators. Function and method decorators were introduced in **PEP 318**.

8.8 Coroutines

Novo na versão 3.5.

8.8.1 Coroutine function definition

```
async_funcdef ::= [decorators] "async" "def" funcname "(" [parameter_list] ")"
                  ["->" expression] ":" suite
```

Execution of Python coroutines can be suspended and resumed at many points (see *coroutine*). In the body of a coroutine, any `await` and `async` identifiers become reserved keywords; *`await`* expressions, *`async for`* and *`async with`* can only be used in coroutine bodies.

Functions defined with `async def` syntax are always coroutine functions, even if they do not contain `await` or `async` keywords.

It is a `SyntaxError` to use `yield from` expressions in `async def` coroutines.

An example of a coroutine function:

```
async def func(param1, param2):
    do_stuff()
    await some_coroutine()
```

8.8.2 The `async for` statement

```
async_for_stmt  ::=  "async" for_stmt
```

An *asynchronous iterable* is able to call asynchronous code in its *iter* implementation, and *asynchronous iterator* can call asynchronous code in its *next* method.

The `async for` statement allows convenient iteration over asynchronous iterators.

The following code:

```
async for TARGET in ITER:
    BLOCK
else:
    BLOCK2
```

Is semantically equivalent to:

```
iter = (ITER)
iter = type(iter).__aiter__(iter)
running = True
while running:
    try:
        TARGET = await type(iter).__anext__(iter)
    except StopAsyncIteration:
        running = False
    else:
        BLOCK
else:
    BLOCK2
```

See also *__aiter__()* and *__anext__()* for details.

It is a `SyntaxError` to use `async for` statement outside of an *async def* function.

8.8.3 The `async with` statement

```
async_with_stmt  ::=  "async" with_stmt
```

An *asynchronous context manager* is a *context manager* that is able to suspend execution in its *enter* and *exit* methods.

The following code:

```
async with EXPR as VAR:
    BLOCK
```

Is semantically equivalent to:

```
mgr = (EXPR)
aexit = type(mgr).__aexit__
aenter = type(mgr).__aenter__(mgr)

VAR = await aenter
try:
    BLOCK
except:
```

(continua na próxima página)

(continuação da página anterior)

```
    if not await aexit(mgr, *sys.exc_info()):
        raise
else:
    await aexit(mgr, None, None, None)
```

See also *__aenter__()* and *__aexit__()* for details.

It is a `SyntaxError` to use `async with` statement outside of an *async def* function.

Ver também:

PEP 492 - Coroutines with async and await syntax The proposal that made coroutines a proper standalone concept in Python, and added supporting syntax.

CAPÍTULO 9

Componentes de Alto-Nível

O interpretador Python pode receber suas entradas de uma quantidade de fontes: de um script passado a ele como entrada padrão ou como um argumento do programa, digitado interativamente, de um arquivo fonte de um módulo, etc. Este capítulo mostra a sintaxe usada nesses casos.

9.1 Programas Python completos

Ainda que uma especificação de linguagem não precise prescrever como o interpretador da linguagem é invocado, é útil ter uma noção de um programa Python completo. Um programa Python completo é executado em um ambiente minimamente inicializado: todos os módulos internos e padrão estão disponíveis, mas nenhum foi inicializado, exceto por `sys` (serviços de sistema diversos), `builtins` (funções internas, exceções e `None`) e `__main__`. O último é usado para fornecer o espaço de nomes global e local para execução de um programa completo.

A sintaxe para um programa Python completo é esta para uma entrada de arquivo, descrita na próxima seção.

O interpretador também pode ser invocado no modo interativo; neste caso, ele não lê e executa um programa completo, mas lê e executa uma instrução (possivelmente composta) por vez. O ambiente inicial é idêntico àquele de um programa completo; cada instrução é executada no espaço de nomes de `__main__`.

Um programa completo pode ser passado ao interpretador de três formas: com a opção de linha de comando `-c` *string*, como um arquivo passado como o primeiro argumento da linha de comando, ou como uma entrada padrão. Se o arquivo ou a entrada padrão é um dispositivo tty, o interpretador entra em modo interativo; caso contrário, ele executa o arquivo como um programa completo.

9.2 Entrada de arquivo

Toda entrada lida de arquivos não-interativos têm a mesma forma:

```
file_input   ::=   (NEWLINE | statement)*
```

Essa sintaxe é usada nas seguintes situações:

- quando analisando um programa Python completo (a partir de um arquivo ou de uma string);
- quando analisando um módulo;
- quando analisando uma string passada à função `exec()`;

9.3 Entrada interativa

A entrada em modo interativo é analisada usando a seguinte gramática:

```
interactive_input   ::=   [stmt_list] NEWLINE | compound_stmt NEWLINE
```

Note que uma instrução composta (de alto-nível) deve ser seguida por uma linha em branco no modo interativo; isso é necessário para ajudar o analisador sintático a detectar o fim da entrada.

9.4 Entrada de expressão

A `eval()` é usada para uma entrada de expressão. Ela ignora espaços à esquerda. O argumento em string para `eval()` deve ter a seguinte forma:

```
eval_input   ::=   expression_list NEWLINE*
```

CAPÍTULO 10

Especificação Completa da Gramática

Esta é a gramática completa do Python, uma vez que é lida pelo gerador de parser e usada para analisar os arquivos contendo código-fonte Python:

```
# Grammar for Python

# NOTE WELL: You should also follow all the steps listed at
# https://devguide.python.org/grammar/

# Start symbols for the grammar:
#       single_input is a single interactive statement;
#       file_input is a module or sequence of commands read from an input file;
#       eval_input is the input for the eval() functions.
# NB: compound_stmt in single_input is followed by extra NEWLINE!
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER

decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
decorators: decorator+
decorated: decorators (classdef | funcdef | async_funcdef)

async_funcdef: ASYNC funcdef
funcdef: 'def' NAME parameters ['->' test] ':' suite

parameters: '(' [typedargslist] ')'
typedargslist: (tfpdef ['=' test] (',' tfpdef ['=' test])* [',' [
        '*' [tfpdef] (',' tfpdef ['=' test])* [',' ['**' tfpdef [',']]]
      | '**' tfpdef [',']]]
  | '*' [tfpdef] (',' tfpdef ['=' test])* [',' ['**' tfpdef [',']]]
  | '**' tfpdef [','])
tfpdef: NAME [':' test]
varargslist: (vfpdef ['=' test] (',' vfpdef ['=' test])* [',' [
        '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
      | '**' vfpdef [',']]]
```

(continua na próxima página)

(continuação da página anterior)

```
  | '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
  | '**' vfpdef [',']
)
vfpdef: NAME

stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | del_stmt | pass_stmt | flow_stmt |
             import_stmt | global_stmt | nonlocal_stmt | assert_stmt)
expr_stmt: testlist_star_expr (annassign | augassign (yield_expr|testlist) |
                     ('=' (yield_expr|testlist_star_expr))*)
annassign: ':' test ['=' test]
testlist_star_expr: (test|star_expr) (',' (test|star_expr))* [',']
augassign: ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' | '&=' | '|=' | '^=' |
            '<<=' | '>>=' | '**=' | '//=')
# For normal and annotated assignments, additional restrictions enforced by the␣
↪interpreter
del_stmt: 'del' exprlist
pass_stmt: 'pass'
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
break_stmt: 'break'
continue_stmt: 'continue'
return_stmt: 'return' [testlist]
yield_stmt: yield_expr
raise_stmt: 'raise' [test ['from' test]]
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS
import_from: ('from' (('.' | '...')* dotted_name | ('.' | '...')+)
              'import' ('*' | '(' import_as_names ')' | import_as_names))
import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',' import_as_name)* [',']
dotted_as_names: dotted_as_name (',' dotted_as_name)*
dotted_name: NAME ('.' NAME)*
global_stmt: 'global' NAME (',' NAME)*
nonlocal_stmt: 'nonlocal' NAME (',' NAME)*
assert_stmt: 'assert' test [',' test]

compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef |␣
↪classdef | decorated | async_stmt
async_stmt: ASYNC (funcdef | with_stmt | for_stmt)
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
while_stmt: 'while' test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
try_stmt: ('try' ':' suite
           ((except_clause ':' suite)+
            ['else' ':' suite]
            ['finally' ':' suite] |
           'finally' ':' suite))
with_stmt: 'with' with_item (',' with_item)*  ':' suite
with_item: test ['as' expr]
# NB compile.c makes sure that the default except clause is last
except_clause: 'except' [test ['as' NAME]]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT

test: or_test ['if' or_test 'else' test] | lambdef
```

(continua na próxima página)

(continuação da página anterior)

```
test_nocond: or_test | lambdef_nocond
lambdef: 'lambda' [varargslist] ':' test
lambdef_nocond: 'lambda' [varargslist] ':' test_nocond
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
# <> isn't actually a valid comparison operator in Python. It's here for the
# sake of a __future__ import described in PEP 401 (which really works :-)
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
star_expr: '*' expr
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
arith_expr: term (('+'|'-') term)*
term: factor (('*'|'@'|'/'|'%'|'//') factor)*
factor: ('+'|'-'|'~') factor | power
power: atom_expr ['**' factor]
atom_expr: [AWAIT] atom trailer*
atom: ('(' [yield_expr|testlist_comp] ')' |
       '[' [testlist_comp] ']' |
       '{' [dictorsetmaker] '}' |
       NAME | NUMBER | STRING+ | '...' | 'None' | 'True' | 'False')
testlist_comp: (test|star_expr) ( comp_for | (',' (test|star_expr))* [','] )
trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
subscriptlist: subscript (',' subscript)* [',']
subscript: test | [test] ':' [test] [sliceop]
sliceop: ':' [test]
exprlist: (expr|star_expr) (',' (expr|star_expr))* [',']
testlist: test (',' test)* [',']
dictorsetmaker: ( ((test ':' test | '**' expr)
                   (comp_for | (',' (test ':' test | '**' expr))* [','])) |
                  ((test | star_expr)
                   (comp_for | (',' (test | star_expr))* [',']) )

classdef: 'class' NAME ['(' [arglist] ')'] ':' suite

arglist: argument (',' argument)*  [',']

# The reason that keywords are test nodes instead of NAME is that using NAME
# results in an ambiguity. ast.c makes sure it's a NAME.
# "test '=' test" is really "keyword '=' test", but we have no such token.
# These need to be in a single rule to avoid grammar that is ambiguous
# to our LL(1) parser. Even though 'test' includes '*expr' in star_expr,
# we explicitly match '*' here, too, to give it proper precedence.
# Illegal combinations and orderings are blocked in ast.c:
# multiple (test comp_for) arguments are blocked; keyword unpackings
# that precede iterable unpackings are blocked; etc.
argument: ( test [comp_for] |
            test '=' test |
            '**' test |
            '*' test )

comp_iter: comp_for | comp_if
comp_for: [ASYNC] 'for' exprlist 'in' or_test [comp_iter]
comp_if: 'if' test_nocond [comp_iter]
```

(continua na próxima página)

(continuação da página anterior)

```
# not used in grammar, but may appear in "node" passed from Parser to Compiler
encoding_decl: NAME

yield_expr: 'yield' [yield_arg]
yield_arg: 'from' test | testlist
```

APÊNDICE A

Glossário

>>> O prompt Python padrão do shell interativo. Muitas vezes visto em exemplos de código que podem ser executados de forma interativa no interpretador.

... O prompt padrão do shell interativo do Python ao se digitar código em um bloco indentado ou dentro de um par de delimitadores direita-esquerda .. XXX: concordam com "delimitadores direita-esquerda"? (como parênteses, colchetes ou chaves).

2to3 Uma ferramenta que tenta converter código Python 2.x para código Python 3.x lidando com a maioria das incompatibilidades que podem ser detectadas analisando o código-fonte e navegando na árvore de sintática

2to3 is available in the standard library as `lib2to3`; a standalone entry point is provided as `Tools/scripts/2to3`. See 2to3-reference.

Classe Base Abstrata Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with *magic methods*). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections.abc` module), numbers (in the `numbers` module), streams (in the `io` module), import finders and loaders (in the `importlib.abc` module). You can create your own ABCs with the `abc` module.

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 :term: *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 *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, o uso do 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**.

assíncrono iterável 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 Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's creator.

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 The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `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**.

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 is a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the *`async def`* statement. See also **PEP 492**.

função coroutine 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)
```

(continua na próxima página)

(continuação da página anterior)

```
@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: *Implementando descritores*.

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 A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statement*s which cannot be used as expressions, such as `if`. Assignments are also statements, not expressions.

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*.

There are actually three categories of file objects: raw *binary files*, buffered *binary files* and *text files*. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

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 *Definições de função* 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 *Definições de função*.

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 An expression that returns an iterator. It looks like a normal expression followed by a *for* expression defining a loop variable, range, and an optional *if* expression. The combined expression generates values for an enclosing function:

```
>>> 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.

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.

All of Python's immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not. Objects which are instances of user-defined classes are hashable by default. They all compare unequal (except with themselves), and their hash value is derived from their `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*.

iterator Um objeto que represent 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 typeiter.

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*.

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 Qualquer classe semelhante a uma tupla cujos elementos indexados também sejam acessíveis por meio de atributos nomeados (como exemplo, tem-se o `time.localtime()` que devolve um objeto semelhante à uma

tupla em que o *ano* é acessível tanto através de um índice, como `t[0]`, quanto por um atributo nomeado como `t.tm_year`).

A named tuple can be a built-in type such as `time.struct_time`, or it can be created with a regular class definition. A full featured named tuple can also be created with the factory function `collections.namedtuple()`. The latter approach automatically provides extra features such as a self-documenting representation like `Employee(name='jones', title='programmer')`.

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 *Definições de 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__
'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 *Nomes de métodos especiais*.

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`.

struct sequence A tuple with named elements. Struct sequences expose an interface similar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `_make()` or `_asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

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 *Annotated assignment statements*.

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 esses documentos

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!

Meus agradecimentos vão para:

- Fred L. Drake, Jr., o criador do primeiro conjunto de ferramentas para documentação do Python e escritor da sua maior parte;
- O projeto Docutils por ter criado reStructuredText e a suíte Docutils;
- Fredrik Lundh por sua Referência Alternativa para Python projeto do qual, Sphinx teve 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.

É somente com o esforço e a contribuição da comunidade Python, que a linguagem possui essa maravilhosa documentação – Obrigado à todos!

APÊNDICE C

História e Licença

C.1 História do software

O Python foi criado no início dos anos 90 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 a 1,2	n/a	1991-1995	CWI	sim
1.3 a 1.5.2	1.2	1995-1999	CNRI	sim
1.6	1.52	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 3.6.15

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   3.6.15 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 3.6.15 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2021 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.6.15 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 3.6.15 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
   3.6.15.

4. PSF is making Python 3.6.15 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 3.6.15 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.6.15
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.6.15, 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 3.6.15, 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 Floating point exception control

The source for the `fpectl` module includes the following notice:

```
  ---------------------------------------------------------------------
 /                      Copyright (c) 1996.                            \
|          The Regents of the University of California.                 |
|                        All rights reserved.                           |
|                                                                       |
|   Permission to use, copy, modify, and distribute this software for   |
|   any purpose without fee is hereby granted, provided that this en-   |
|   tire 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  work was produced at the University of California, Lawrence   |
|   Livermore National Laboratory under  contract  no.  W-7405-ENG-48   |
|   between  the  U.S.  Department  of  Energy and The Regents of the   |
|   University of California for the operation of UC LLNL.              |
|                                                                       |
|                              DISCLAIMER                               |
|                                                                       |
|   This  software was prepared as an account of work sponsored by an   |
|   agency of the United States Government. Neither the United States   |
|   Government  nor the University of California nor any of their em-   |
|   ployees, makes any warranty, express or implied, or  assumes  any   |
|   liability  or  responsibility  for the accuracy, completeness, or   |
|   usefulness of any information,  apparatus,  product,  or  process   |
|   disclosed,   or  represents  that  its  use  would  not  infringe   |
|   privately-owned rights. Reference herein to any specific  commer-   |
|   cial  products,  process,  or  service  by trade name, trademark,   |
|   manufacturer, or otherwise, does not  necessarily  constitute  or   |
|   imply  its endorsement, recommendation, or favoring by the United   |
|   States Government or the University of California. The views  and   |
|   opinions  of authors expressed herein do not necessarily state or   |
|   reflect those of the United States Government or  the  University   |
|   of  California,  and shall not be used for advertising or product   |
 \  endorsement purposes.                                              /
  ---------------------------------------------------------------------
```

C.3.4 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.3.5 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.

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.3.6 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.3.7 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
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
```

(continua na próxima página)

(continuação da página anterior)

```
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.8 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.3.9 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
```

(continua na próxima página)

(continuação da página anterior)

```
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.3.10 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.3.11 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski' implementation of Dan Bernstein's SipHash24 algorithm. The contains the following note:

```
<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>
```

(continua na próxima página)

(continuação da página anterior)

```
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.3.12 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.3.13 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
 ---------------

   /* ====================================================================
```

(continua na próxima página)

(continuação da página anterior)

```
    * 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
    * 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)
```

(continua na próxima página)

(continuação da página anterior)

```
   * 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
   * 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.3.14 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.3.15 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.3.16 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.3.17 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
```

(continua na próxima página)

(continuação da página anterior)

```
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.3.18 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.

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.

Índice

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